Using Argumentation to Control Lexical Choice: A Functional Unification Implementation

Michael Elhadad

Submitted in partial fulfillment of the
requirements of the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences
Columbia University
1992

ABSTRACT

Using Argumentation
to Control Lexical Choice:
A Functional Unification Implementation

Michael Elhadad

This thesis investigates the impact of the pragmatic situation on surface generation. It presents new surface generation techniques that improve on both aspects of surface generation: (1) lexical choice, which consists of choosing words and their associated syntactic structures and (2) syntactic realization, which consists of combining these partial structures into grammatical sentences. Because surface generation depends directly on aspects of the pragmatic situation, these new techniques allow a purely conceptual input to be expressed by a greater variety of linguistic forms and with more sensitivity to pragmatic factors than was previously possible.

Specifically, this research focuses on the impact on lexical choice of one part of the pragmatic situation: the speaker's argumentative intent, *i.e.*, the goal of the speaker to convince the hearer of a certain conclusion. The argumentative intent can be realized by a variety of evaluative expressions appearing at various ranks in the syntactic structure. This thesis describes the selection of four classes of evaluative expressions: judgment determiners (*i.e., many*), scalar adjectives (*i.e., difficult*), connotative verbs (*i.e., require, enjoy*) and argumentative connectives (*i.e., but, so*). These four classes were not addressed in previous generators.

The FUF formalism is introduced in this thesis to address the issue of complex constraint interaction arising when performing lexical choice under pragmatic constraints. FUF is derived from Functional Unification Grammars (FUGs), a formalism previously used for syntactic realization only. FUF extends on FUGs by providing new mechanisms for control, for expressing hierarchical relations and for using modular knowledge sources. These extensions make FUF capable of handling lexical choice. In addition, this thesis describes the use of FUF to develop SURGE, the largest and most widely used syntactic realization grammar available.

Finally these new generation techniques are applied to the implementation of ADVISOR II, a question-answering system helping students to choose classes for a semester. In particular, ADVISOR II uses the increased expressive flexibility provided by these techniques to convince a student to choose different classes by presenting the same objective data, retrieved from an underlying knowledge base, in different linguistic forms using evaluative expressions.

Acknowledgements

I am very happy to complete this thesis and it is my great pleasure to thank all those who have helped me work on it and those who made my life enjoyable and interesting during the period I worked on it.

First, Kathleen R. McKeown has introduced me to the field of generation, patiently taught me how to write and encouraged me in all possible ways. She has also constructed an environment at Columbia in which fun research is possible. Without her constant support, none of this work would have been possible.

My friends Jacques Robin, Frank Smadja and David Kurlander have formed the core of my social network during my years at Columbia. Jacques X. Robin has had an enormous influence on this work. His ideologic/ayatollic approach to research has forced me to straighten up my arguments. His systematic testing has forced me to debug my code. His cynical enthusiasm has kept me focused. But overall, Jacques has served as my research psychiatrist - always ready to spend days hearing me mumble some theoretical fantasies and making it sound interesting to me. Frank Z. Smadja's healthy paranoia has also made life more interesting. His admirable efficiency has helped balance my tendencies to wander in all directions. David J. Kurlander, on the other hand, has made these tendencies grow ever always stronger - and has helped me develop tremendous thesis avoidance strategies (which have served him well as well). David's excellent Chimera graphical editor has created the most remarkable diagram of this thesis.

Thanks to Doree D. Seligmann for banging on the door whenever she passed, to Ari Gross for teaching me Mishnayot, to Jason Glazier for sharing the excitement of Wall Street trading, to Tony Weida for telling me about his bike trips and to Becky Passonneau for teaching me linguistics.

My brother Salomon and all of the Dalet team - David, Stephane and Charles - have made me enjoy the meteoric quality of business life. I want to thank my parents for the model they have always provided and the strength they have inspired me.

Life in New York would have been hard without the friendship of Rabbi Kret and the warmth of Old Broadway. Thanks also to my good friends Josh and Raizy Zakheim, Dan and Rina Yellin and to Dan and Esther Victor for cool all American Thanksgiving days. Thanks to Leslie and Bill for moving up with us.

Life would have been hard anywhere without Catherine. Thanks for being here. During the time it took to complete this work, our daughters Deborah and Hannah were born, grew up and mastered 2 and a half languages bidirectionally. This certainly put things into perspective. And it made me very happy.

This work was supported by grants from the National Science Foundation under Grant IRT-84-51438, the New York State Center for Advanced Technology under Contract NYSSTF-CAT(88)-5, and the Office of Naval Research under Contracts N00014-89-J-1782, DARPA under contract N00039-84-C-0165, GE and Bellcore.

To Catherine

To my Parents

To little Miriam, who will be very happy

Chapter 1
Introduction

This thesis presents new techniques in lexical choice that allow for the generation of more fluent text then was previously possible. Lexical choice is defined here as the interface in a generation program between a conceptual representation expressing what information is to be conveyed by the text, and a linguistic representation, where all open class lexical items are selected and lexical dependencies are specified. Three aspects of this problem are covered in this work:

- First, the focus is put on a specific class of lexical decisions, the use of evaluative expressions to satisfy a speaker's argumentative intent. Two characteristics of this class of decisions highlight important features of the lexical choice task in general: (1) choosing an evaluative expression is a lexical decision which *directly* corresponds to a pragmatic goal and (2) the same argumentative intent can be realized by a variety of lexical items at different syntactic ranks. In contrast, existing surface generators either assume that (1) lexical decisions only depend on the propositional content to be conveyed in an utterance and cannot interact directly with content selection or (2) a given input constraint can be realized only at a single syntactic rank.
- Second, because the task of the lexical chooser is more complex when these two features are taken into account, a new formalism capable of addressing this complexity was developed. Two factors contribute to the added complexity of lexicalization: (1) more pragmatic factors are taken into account and, consequently, decisions can interact in complex ways, and (2) because many lexical items at different syntactic levels can satisfy the same pragmatic goal, the search space for an appropriate linguistic structure is much larger. This thesis presents the FUF generation formalism. FUF is a declarative formalism derived from the Functional Unification Grammars formalism which I have used to implement both syntactic realization and lexical choice.
- Finally, because certain lexical decisions affect syntactic realization, a syntactic realization grammar of English was developed. This thesis presents the SURGE grammar, a large, robust syntactic realization grammar of English written in FUF.

Evaluation Expressions

I specifically address the issue of generating *evaluative expressions* to satisfy an *argumentative intent, i.e.*, the communicative goal of convincing a hearer of a certain conclusion. To illustrate this issue, I have developed ADVISOR II, a program which advises university students on which courses to take for a semester. The interaction shown in Fig.1-1 is produced by ADVISOR II for a certain student profile.[1] The argumentative intent realized by this paragraph is to convince the student not to take the AI class. Evaluative expressions in the paragraph produced by the system are show in bold. They include judgment determiners (*many*), scalar adjectives (*theoretical* and *difficult*), argumentative connectives (*so*) and connotative verbs (*require*). Such expressions are pervasive in natural language, and their generation has been the object of surprisingly little work.

Studying how an argumentative intent is realized allows one to focus on two features of lexical choice: (1) the argumentative intent directly impacts not only the choice of content, but also the selection of evaluative expressions. As a consequence, surface realization and content selection can interact in often complex ways; (2) evaluative expressions are varied and can appear at various syntactic levels. The same argumentative intent can be realized by

[1]This paragraph is produced when the system assumes that the student has little background in theory.

Q: Should I take AI?

A: AI covers logic, a **very theoretical** topic,
and it **requires many** assignments.
So it could be **difficult**.
I would not recommend it.

Figure 1-1: A paragraph with evaluative expressions

linguistic elements at different levels in the linguistic structures. I call such input constraints on the generation process *floating constraints* because they can ''float'' in the syntactic structure. The lexical chooser presented in this thesis is capable (1) of interacting with the content selection module of the generator and (2) of handling floating constraints in the input.

Formalism

These new features significantly increase the flexibility of the generator. They also create new computational challenges, because controlling the mapping between conceptual and linguistic structures becomes a highly non-deterministic task when floating constraints are considered. I have extended the formalism of Functional Unification Grammars (FUGs) into a system called FUF to deal with this increased complexity. FUF is a programming environment specialized for the development of text generation systems and is a second focus of this work.

FUF is a declarative formalism for surface realization that is powerful enough to implement lexical choice techniques and improve on syntactic realization implementations. The main problem with competing surface generation formalisms (MUMBLE [Meteer et al. 87] and NIGEL [Mann & Matthiessen 83a]) is that they are deterministic. Consequently, when using these formalisms, all the generation decisions must be specified by the system developer in the order in which they will be executed by the program. There is, however, no single order of decisions which will account for all pragmatic situations and input configurations. The original FUG formalism was found to be superior to these other formalisms to address lexical choice issues because it is non-deterministic and, therefore, allows the order of constraints specification to be dynamically determined. But FUGs were not sufficient for addressing all aspects of lexical choice, especially to deal with floating constraints. A contribution of this thesis is to identify configurations of input constraints which were difficult to solve using the original FUG formalism and to develop an extended formalism making the solution of such configurations possible and efficient.

FUF is derived from the Functional Unification Grammars (FUGs) formalism. FUGs have been successfully used for syntactic realization in previous work [McKeown 85; Appelt 85; Paris 87]. This thesis presents the first application of this formalism to the task of lexical choice. FUF includes all the facilities of the original FUG formalism with the following extensions:

- New control mechanisms allowing the implementation of larger and more complex grammars and the efficient processing of floating constraints.
- Support for types and inheritance, making the formalism more expressive and concise.
- Support for modular grammars and interaction with external knowledge sources.

Grammar

Finally, the third aspect of this work is the development of SURGE a large, robust syntactic realization grammar written in FUF. SURGE is a portable system, with one of the largest coverage among generation grammars for English. SURGE is mainly based on a systemic description of English. It has been distributed to other research teams and has become the most widely used surface generation grammar at this time.

SURGE processes the output of the lexical chooser, to combine the entries found in the lexicon into grammatical structures. The developments of the lexical chooser and of SURGE are therefore interdependent. In some cases, the lexicon encodes most of the decisions. This is the case for adjectives. In other cases, the grammar encodes most the decisions. This is the case for the determiner sequence.

Main contribution

In summary, the three main contributions of this work are:

- The development of a lexical chooser capable of (1) generating evaluative expressions, (2) interacting with the content selection module of a generator, and (3) handling floating constraints.
- The development of FUF, a new efficient formalism capable of handling both lexical choice and syntactic realization.
- The development of SURGE, a large, portable syntactic realization grammar written in FUF.

In the rest of this chapter, I first elaborate on the motivation for this work. I then describe the starting points and questions on which this thesis builds. The range of techniques presented in the thesis is then described in the context of the ADVISOR II system. The contributions of the thesis are summarized and the introduction ends with a guide to the rest of the dissertation.

1.1. Motivation

This work seeks to attain the following specific goals:

- **Provide the ability to produce evaluative text**: evaluations are pervasive in natural text, yet little attention has been paid to the problem of generating evaluations in a principled manner.
- **Increase paraphrasing power**: the surface realization component should be capable of generating a large variety of linguistic forms corresponding to the same input information. This implies that the input to the surface generation module must be general enough to account for many different syntactic and lexical decisions. When this is the case, the generator is capable of producing various forms adapted to different pragmatic situations.
- **Develop a computational formalism appropriate for text generation**: the identification of new tasks for surface realization has also led to the identification of new needs in the programming support generation requires. I have worked on the FUF formalism to address these needs and better understand what facilities a generation-oriented formalism should provide.

This section explains why the issue of generating evaluative expressions provides a fertile testbed to understand the place of lexical choice within the generation process and the challenges it poses.

1.1.1. Generate Argumentative Evaluation

The first goal of this work is to systematically study how argumentative evaluations can be realized linguistically. Argumentative evaluations correspond to a speaker's goal to convince the hearer of a certain conclusion or to perform a certain action. The ability to reason about evaluations and their linguistic realizations endows a generator with the following capabilities:

- **Ability to express non-objective information**: that is, with a conceptual description denoting the same information as input, the generator can produce different linguistic realizations. For example, from a description specifying that AI has seven assignments, the generator can produce *AI requires many assignments* or *AI has seven assignments*, depending on its argumentative intent. The generator can also produce purely evaluative sentences, such as *AI is difficult* or *AI is a good course*, which are situation-dependent judgments, and therefore cannot be directly derived from information stored in a knowledge base.
- **Ability to produce hearer-specific text**: that is, with the same information as input, but with different assumptions about the hearer, the generator can produce different linguistic realizations. For example, if the hearer is believed to enjoy programming, then the system can generate *AI has a lot of programming* from the description of the AI assignments. In contrast, if the hearer is believed to be indifferent to programming but inexperienced in writing essays, then the generator would not include

any information about programming, and the same description of the AI assignments would lead to the realization *AI has many essay assignments*.

- **Ability to produce goal-sensitive text**: that is, with the same information as input, the same user description, but with a different argumentative goal, the generator can produce different linguistic realizations. For example, consider a situation where the following user model assumptions hold: the student is interested in AI, has experience in programming, little experience in mathematics and no experience writing papers. If the system is set to convince the hearer that AI is a good class, then the following paragraphs can be generated:

  > AI has many programming homeworks and it covers a lot of interesting topics, such as NLP, Vision and KR. So it should be quite interesting. I would recommend it.
  >
  > AI covers logic, a very theoretical topic, so it could be difficult. But it has many interesting topics, such as NLP, Vision and KR. It should be a good class.

  In contrast, if the system is set to convince the hearer not to take AI, then different paragraphs can be generated, even while using the same user model:

  > AI covers logic, a very theoretical topic and it requires many paper assignments. You have little experience writing papers. Therefore it could be quite difficult. I would not recommend it.
  >
  > AI covers many interesting topics, NLP, vision and KR, but it deals with logic, a very theoretical topic. So it could be difficult. I would not recommend it.

In terms of surface realization and lexical choice, the study of evaluative expressions increases the coverage of the generator by providing reasons to select linguistic devices such as scalar adjectives, judgment determiners and argumentative connectives, whose function is primarily to express evaluations. Such expressions had not been studied in previous work in surface generation.

Focusing on evaluative expressions also highlights two important features of the lexicalization process. Figure 1-1 (p.2) illustrates the first one: when mapping an argumentative intent onto text, the argumentative intent impacts not only what information is presented to the hearer (content selection), but also the surface form of the text (surface realization). Thus, for example, the fact that AI has seven assignments is selected as relevant information because it can serve as an argument for the conclusion that AI is difficult and, therefore, not a good class for the hearer. The same argumentative intent determines the lexical decision to use *many* instead of *seven*, or to use any of the other evaluative expressions highlighted in Fig.1-1. While previous work in natural language generation has concentrated on techniques for either content selection (deciding *what to say*) or for surface realization (deciding *how to say it*) separately, little attention has been paid to issues that arise when generating evaluative expressions, and which affect both aspects of the generation process simultaneously. The main issue here is that content selection and surface realization decisions can interact in often complex ways. This thesis presents an architecture for a text generator where content selection and surface realization are constrained by an input argumentative intent in a uniform manner. This approach addresses a question raised some time ago in [Danlos 87a]: how can surface realization decisions interact with content selection decisions?

A second feature of the task of satisfying an argumentative intent is illustrated in Fig.1-2. All four sentences in the figure convey similar information and satisfy the same argumentative intent: they evaluate the AI class as high on the scale of difficulty. The difference is that this evaluation is realized at four distinct syntactic ranks.

- (1) Judgment determiner: *AI has many assignments.*
- (2) Predicative scalar adjective: *AI is difficult.*
- (3) Connotative verb: *AI requires seven assignments.*
- (4) Argumentative connective: *AI is interesting but it has seven assignments.*

Figure 1-2: An evaluation expressed at different syntactic ranks

In (1), the evaluation is realized by selecting the judgment determiner *many* and relying on the inference rule *the more assignments in a class, the more difficult it is*. Here the lexical chooser decided to use the marked evaluative expression *many* instead of *seven*. Judgment determiners include *many*, *few*, *a great number of* etc.

In (2), the use of a scalar adjective directly realizes the evaluative intention of the speaker. Scalar adjectives include *difficult, interesting, important* etc.

In (3), the choice of the main verb *require* as opposed to *AI has seven assignments* can also be related to the speaker's intention to evaluate AI as a difficult class. The verb *require* fulfills two functions: it expresses the relation between the class and the assignments, and it also expresses the connotation that AI is difficult. In contrast, the possessive relation *AI has assignments* only fulfills the first function and does not have the same connotation. Connotative verbs can be used to merge two communicative goals into a single linguistic construct.

Finally, in (4), the argumentative connective *but* projects an evaluation on the clauses it connects in an indirect way. The clause *AI has seven assignments* is not evaluative taken alone; but when it is contrasted to the first evaluation *AI is interesting*, it becomes an argument that must be opposed to *interesting*, and the whole sentence supports this second argument. In this case, the speaker relies on two common sense relations that predict that (a) the more a course is interesting the more a student wants to take it and (b) the more a course is difficult, the less a student wants to take it. Such common sense relations are called *topoi* in [Anscombre & Ducrot 83]. Topoi play an important role in the techniques presented in this thesis.

The communicative goal to evaluate an entity on a *scale*, for example, to evaluate the class of AI on the scale of difficulty, can be realized by very different linguistic devices. The four constructions shown in Fig.1-2 operate at distinct syntactic levels (connective, main verb, noun modifier, determiner sequence). The lexical choice method presented in this thesis is, therefore, capable of mapping the same input onto a variety of linguistic constructions. I call this feature *cross-ranking realization*. Input constraints which, like evaluations, can be realized at different syntactic levels are called *floating constraints*. The need to perform cross-ranking realization and to deal with floating constraints requires that the input to the generator be neutral to linguistic form. This is in sharp contrast with existing generators [Bateman et al. 90; Meteer et al. 87] whose input already determines linguistic structure (*e.g.*, a semantic predication is realized as a clause, individuals are realized as noun phrases). The distinction between the structure of the conceptual input and the linguistic structure used to realize it implies that the lexical chooser must not only perform paradigmatic choices (select among substitutable items, *e.g.*, between *require* and *necessitate*), but also structural choices (determine the linguistic structure corresponding to a given input specification, *e.g.*, select between a clause and a nominalization, determine which construction will realize an evaluation). In previous work, these structural decisions were made implicitly at the content determination stage, when building the input to a text generator, and usually through hand-coding. In this thesis, they are made by the realization component.

In summary, taking into account argumentative intents during generation requires addressing two questions:

- How can a pragmatic goal directly affect all generation decisions, from content determination to lexical choice and syntactic realization.
- How can the same input constraint be realized by a variety of lexical and syntactic resources, *i.e.*, how can a generator perform cross-ranking realization on floating contraints.

These two questions were not addressed in previous work, and as a consequence the following limitations were present in existing text generators:

- Input to existing surface generators is too tightly coupled to linguistic form. As a consequence, cross-ranking realization is not possible, and floating constraints are not considered. This over-determination of the input severly limits the paraphrasing power of a generator.
- Existing surface generators are not sensitive enough to the pragmatic situation and therefore cannot adapt the text they produce to a variety of situations in a principled manner.[2]

[2]With the exception of PAULINE [Hovy 88a] which concentrates on this problem.

1.1.2. Extend the Generator's Paraphrasing Power

The second goal of this work is to increase the generator's paraphrasing power, *i.e.*, provide the capability to generate many different surface forms for the same input information. Paraphrasing is important because (1) it increases variety in the generator's output and (2) when the choice between paraphrases can be related to pragmatic factors, it increases the generator's sensitivity to the pragmatic situation. When paraphrasing is limited, in certain situations a generator can simply be unable to produce text.

Existing surface generators often have limited paraphrasing power because, as noted above, their input is already committed to certain linguistic decisions. For example, sentence generators typically expect their input to be in the form predicate(arg1, arg2, arg3). In such a case, the predicate is always realized by the verb of the sentence, and each argument is realized by one syntactic complement of the verb (subject, object, indirect object). In this case, the linguistic structure is isomorphic to the structure of the ''semantic'' input. When dealing with argumentative evaluations, this approach cannot work. The reason is that the argumentative evaluation is a *floating constraint* which can be realized at different syntactic levels. In the example shown in Fig.1-2, the same argumentative evaluation was realized at the determiner level in (1), attributive complement in (2), main verb in (3) and connective in (4). The same input constraint can therefore float in the linguistic structure: the two structures, linguistic and conceptual, are therefore not isomorphic. Evaluations are not the only case of floating constraints. Danlos has studied the expression of causality and her findings support the conclusion that causality is also a floating constraint, although she does not use this terminology [Danlos 87a].

To allow the generator to account for such cases of non-isomorphism, the input must not be committed to any a priori linguistic decisions and be general enough to represent all paraphrases. When considering such input, new issues arise in lexical choice and syntactic realization. Among these, I have focused on the issues of clause and NP planning. In both cases, the issue is to map conceptual modifiers of an entity onto a syntactic structure.

At the clause level, *clause planning* includes the selection of a head relation among several competing conceptual relations. By selecting different heads the same conceptual representation, the following sentences can be generated:[3]

- *Intro to AI requires two assignments which require writing essays, in which you do not have experience.*
- *Two assignments of Intro to AI require writing essays, in which you do not have experience.*
- *You do not have experience in writing essays which two assignments of Intro to AI require.*

When the input to the surface realization module is neutral with respect to linguistic structure, the lexical chooser must perform clause planning and choose among these alternatives.

Similarly, at the NP level, a large variety of linguistic structures can be produced to describe the same information. For example, consider the set of topics containing the three elements *expert-systems*, *knowledge representation* and *vision* and defined in intension as the set of topics covered in AI such that the user is known to be interested in them. Then, depending on which aspect of this definition needs to be highlighted the following lexicalizations illustrate the variety of forms that can be generated:

- *Many interesting topics covered in AI, such as Vision, Expert Systems and Knowledge Representation.*
- *Vision, Expert Systems and Knowledge Representation.*
- *Three AI topics.*
- *Interesting topics.*
- *Three interesting topics.*
- *A large number of AI topics that you should find interesting.*

Here again, the task of NP planning, *i.e.*, selecting one of these forms from the common conceptual representations they realize, must be performed by the lexical chooser.

---

[3]Note that, certain of these sentences are not felicitous, based on style considerations. But the different syntactic structures are all grammatical.

A lexical chooser capable of freely selecting a perspective on a complex conceptual network and of lexicalizing complex NPs is much more flexible than existing generators. This flexibility significantly increases the paraphrasing power of the generator and the fluency of the generated text. More importantly, the pragmatic effect of these different paraphrases is different, and the ability to choose a different clause or NP structure to satisfy a pragmatic goal means that the system can choose an expression that is right given the current situation.

1.1.3. Extend Formalism for Text Generation

The third goal of this work is to develop a robust formalism specifically designed to implement text generation applications. A formalism that provides facilities to develop practical generation applications is critical to make generation more accessible as a technology. I have developed FUF, an extension of the Functional Unification Grammar (FUG) formalism to fulfil this need. Two goals have directed my work in this area:

- **Expressiveness**: existing formalisms used for surface realization (mainly, MUMBLE [Meteer et al. 87] and NIGEL [Mann & Matthiessen 83a]) impose stringent restrictions on the generation process: both formalisms are deterministic, requiring the grammar writer to order the decisions that must be made during generation in a fixed manner; they are also procedural and uni-directional, thus forcing the grammar writer to describe *how* each constraint must be enforced and how it interacts with other constraints, instead of just specifying what constraint must be enforced. In contrast, FUF is non-deterministic, declarative and bi-directional. These features allow the grammar writer to specify each constraint separately, and to let the formalism find a way to find a configuration of features satisfying all input constraints.
- **Usability**: a formalism must be computationally efficient enough to be usable. It must also provide facilities for the development of large grammars.

1.2. Starting Points

The work described in this thesis centers around a core set of issues in the field of generation. Among the most important addressed in this work are the place of lexical choice in the generation process, the role of argumentation in generation, the identification of difficult tasks for a realization component and the definition of the role of a formalism for text generation.

1.2.1. Lexical Choice in the Generation Process

Lexical choice is the process of selecting open-class lexical items in a linguistic structure. The place of lexical choice in the generation process is still controversial in the field. A large part of the generation community seems to agree that the task of syntactic realization can be fairly well isolated from the rest of the generation process. As evidence of this position, portable syntactic realization components such as SURGE, NIGEL [Mann & Matthiessen 83a] and MUMBLE [Meteer et al. 87] have been developed, which all expect as input a fully lexicalized specification. This view implies that lexical choice can be performed before syntactic realization starts.

In contrast, some researchers, who often work on reversible grammar formalisms - that is, people interested in using the same grammar to both parse and generate language - advocate performing lexical choice within the syntactic realization stage, viewing lexical choice as just a very specific type of syntactic decision (cf. for example [Shieber et al 90] and [Van Noord 90]). The range of constraints on lexical choice covered in this work is, however, quite restricted.

I take in this work the first view that lexical choice and syntactic realization can be clearly separated and implemented as two successive modules in the architecture of a generation system. I will show here that in such an architecture all lexical decisions can be made before syntactic realization starts. Since this separation makes implementation much easier and reduces complexity, I have found no reason to forgo its advantages.

The second aspect of this issue is to determine the place of lexical choice with respect to content determination.

Certain systems view lexical choice as part of the stage where a generator decides what to say. This is the position held, for example, in [Danlos 87a], [Kukich 83a] and [Jacobs 85]. In this view, lexical resources (represented as phrasal entries) determine what can be said of a domain in a strong sense.

In contrast, many researchers view lexical choice as a distinct operation, that can be performed after all content has been determined and expressed in a language-independent manner. This is the approach held, for example, in TEXT [McKeown 85], SPOKESMAN [Meteer 89] and COMET [McKeown et al 90]. One problem with this approach is that it imposes constraints on the content description language: it must be possible to realize any content specification in language, and the content specification must be structured in such a way that each chunk can be realized by a single linguistic constituent. This means that the content determination module must be aware of the linguistic system to at least segment the conceptual material in chunks of the right size. This problem has been named the *expressibility problem* in [Meteer 90]. It is also addressed in [Rubinoff 92].

In this work, I have also come to the conclusion that the expressibility problem is critical in the design of a generation system. In particular, as the lexical chooser becomes more flexible, the size of the conceptual chunks that can be realized by a single clause varies accordingly. I address this problem by taking the following stance:

- The content representation language should be as language independent as possible. In particular, the structure of the conceptual input should not be committed to any specific linguistic structure. For example, a conceptual relation can be realized by a clause, a nominalization, a noun-noun modifier, a predicative adjective or a prepositional phrase. Another aspect of the mismatch between conceptual and linguistic structures is that several conceptual elements can be mapped onto a single linguistic constituent, using devices such as composite processes and connotative verbs.
- The content organization language must be flexible enough to accomodate building blocks of variable sizes.

The decoupling of conceptual structure and linguistic structure is an important premise of this work. Relying on a conceptual representation neutral with respect to the linguistic structure partially addresses the problem of expressibility, but its main benefit is that it also allows for more paraphrasing power (including structural paraphrasing) in the lexical chooser.

1.2.2. The Role of Argumentation in Generation

This work focuses on the realization of argumentative evaluations. The notion of argumentation used in this work is derived from the theory of Argumentation in Language presented in [Anscombre & Ducrot 83]. The starting point of this theory is that within the semantics of lexical items, one must find indications of how the items contribute to the argumentative orientation of a sentence. The assumption is that sentences, when uttered in a certain situation, are used to present arguments for or against specific conclusions. Their theory hypothesizes that arguments can be linked to conclusions by using commonsense relations called *topoi*. Topoi are gradual inference rules of the form *the more X is P, the more Y is Q, e.g., the more assignments in a class, the more difficult the class*. Entries in the lexicon specify which class of topoi can be activated when a lexical item is used in a sentence.

The work presented in this thesis is the first application of this notion of argumentation to the field of generation. Work in the theory of argumentation has concentrated on the definition of argumentative connectives and operators (*e.g., but, however*). In this work, I extend the scope of the theory and apply it to verbs and adjectives, but more importantly, look at the compositionality of argumentative indications. That is, I examine how the use of a judgment determiner embedded in an NP can trigger the activation of a topos applied to the whole clause.

I have also found that argumentation can play an important role in determining the effect of content organization on surface realization. An often underestimated fact is that the position of a clause within a text structure affects the way the clause is verbalized.

A very simple example of this effect is provided when considering argumentative text. Consider two text plans, where the same information - the number of assignments in a course - is used as an argument to either take the course or avoid it. In the first case, the information is realized as *AI requires many assignments*, in the second case, the same information is realized as *AI has a few assignments*. In this example, the connection between the number of assignments and the desire to take a class is used both to organize the text and to affect its surface realization.

In general, I will show how argumentation and topoi, in particular, can serve as a bridge between content organization and surface realization. Topoi are used both to select and organize content and to perform lexical choice at the clause level. This point is elaborated in Chap.2.

The work that is most closely related to the goal of determining surface form based on the speaker's intent is PAULINE [Hovy 88a]. In PAULINE, stylistic goals determined all aspects of the generation task, from content determination to lexical choice.

1.2.3. Cross-Ranking and Floating Semantic Elements

Generation has been defined as the process of making choices under constraints [McDonald 80]. An important issue for the field of generation is to identify which decisions are difficult and why. In other words, what makes generation a challenging task? Focusing on the surface realization task (as opposed to text planning), the task of sentence-level planning has emerged as one of the most challenging within the realization process.

Sentence-level planning consists of mapping semantic elements onto a linguistic structure, deciding, for example, that a semantic predication is to be realized by a nominalization, a noun phrase modifier or by the main clause. I have characterized this problem by identifying a class of semantic elements that can be realized by linguistic constituents at several ranks within the syntactic structure. I have called such elements *floating semantic elements*. Floating elements pose a problem to a generator because they highlight the mismatch between the input conceptual structure and the output linguistic structure. This problem has been identified by Talmy, in a linguistic analysis of lexicalization patterns, but little consideration has been given to it in generation:

> A combination of semantic elements can be expressed by a single surface element, or a single semantic element by a combination of surface elements. [Talmy 85, p.57]

I therefore identify the problem of *cross-ranking realization* - that is, the ability to realize the same semantic element at different syntactic ranks - as a central issue in surface realization and propose a solution to this problem based on three elements: (1) the input to a surface generator must be general enough to allow for cross-ranking realization; (2) the choice between different syntactic ranks is determined by the interaction of several constraints; finally (3) a functional unification implementation is well adapted to handling the problem of cross-ranking when annotations are added to the grammar and lexicon to explicitly indicate what are the potential sites of realization of each floating constraint. I will show in the thesis that this type of grammatical knowledge, typically not captured in computational grammars, is critical to making the processing of floating constraints efficient and understandable.

1.2.4. Formalism for Text Generation

The definition of a specialized formalism for developing text generation applications has ranked high on the list of goals of this work. For surface realization, four types of formalisms have been used:

- Psychologically motivated formalisms include MUMBLE [Meteer et al. 87] and IPG [De Smedt 90]. These formalisms seek to simulate a psychologically valid model of speech production. In particular, they are deterministic (no decision is ever retracted) and incremental (the output is produced piece by piece in left-to-right order).
- Linguistically motivated formalisms include NIGEL [Mann & Matthiessen 83a], which implements the systemic functional theory of linguistics and [Polguere 90] which implements Meaning-Text Theory.
- Procedural implementations are basically computer programs which are not motivated by other goals than to ''work''.
- Reversible formalisms are motivated by the goal to use the same declarative formalism to both parse and generate language. Reversible formalisms are often based on logic programming [Shieber et al 90; Van Noord 90]. FUG is also by nature a reversible formalism, although in the use of FUGs made in this work, no attention has been paid to this feature.

In this work, reversibility and psychological validity have not been considered as goals. The FUF formalism has

been developed to implement linguistically well-founded grammars and with the goal to be a useful and convenient programming environment to make generation systems work.

Several other issues characterize the work on formalism presented in this thesis:

- **Expressiveness vs. worst-case complexity**: there is a trade-off between the expressiveness of a formalism and its worst-case complexity. I have opted in this work to use and develop an expressive formalism, without concern for worst-case complexity, as long as the typical complexity does not reach the worst-case limits. For example, the FUF formalism is Turing-equivalent but the actual complexity of using a FUF grammar depends on the non-determinism encoded in the grammar and on the degree of instantiation of the input specification. In practice, FUF is an efficient formalism, and generation using SURGE takes less than two seconds real time on a 1991-class workstation (28MIPS) for a single sentence.
- **Uniformity**: I have tried to use the same formalism for as many generation tasks as possible. In particular, this work is the first attempt at using a FUG-approach to perform lexical choice and part of content determination beyond syntactic realization, which was the only task for which FUGs used to be applied. A motivation for this uniformity was the work on argumentation and the use of topoi as a bridge between content determination and surface realization. Since the same data-structure (topoi) can be used to influence the two tasks, it is desirable to use the same formalism to encode both tasks. A uniform formalism also allows for an easier specification of interactions between the two tasks. This stance is in contrast to the position advocated in [McDonald et al. 87] that separate tasks should be implemented using distinct specialized formalisms.
- **Usability and formal soundness**: Formal analyses of feature structure systems similar to FUF have been established in [Ait-Kaci 84], [Kasper 87] and [Kasper&Rounds 86]. The work presented in this thesis is informed by this formal analysis and I have ensured that most extensions added to the base FUG formalism were compatible with these formal results. In particular, FUF is monotonic and bidirectional. I have, however, adopted a pragmatic approach towards the formalism and introduced impure features to the formalism to boost its usability, *i.e.*, features which are not monotonic or bidirectional. This approach is similar in spirit to the introduction of non-logical arithmetic facilities and of the cut construct in Prolog. While these features do not fit well within the semantics of logic programming, they make the language more usable. Similarly, I have introduced features such as `given` and `test` which do not fit well in the logical semantics of feature structures, but make the FUF formalism more practical.

1.3. Implementation

In this section I present the ADVISOR II system, a practical application of the new generation techniques developed in this thesis. I discuss both the type of architecture and the type of conceptual input required by a generator using the lexical choice techniques introduced in this thesis. I start by describing the role of argumentation in the interaction between lexical choice and content planning. I then briefly discuss the interaction between lexical choice and syntactic realization using SURGE.

The ADVISOR II system is a question-answering system assisting students planning their schedule for a semester. It is a variant of the ADVISOR system developed at Columbia in 1982-1986 [McKeown 88]. ADVISOR II takes as input a question of the form *how <scale> is <class>*, for example, *how difficult is AI* or of the form *should I take <class>*. The output is an evaluation of the class tailored to a particular user model. Figure 1-3 is an example of output generated by the system. The two paragraphs are generated with the same user model, specifying that the user is interested in AI and programming and has little experience in writing papers and in theory.

Figure 1-4 outlines the architecture of the system. I have concentrated on three modules in this architecture: the evaluation system, the lexical chooser and the surface realization grammar.

The main focus of this work is the generation of evaluative expressions. Such expressions are characterized by two features: (1) they are user specific, and (2) they are not fully determined by objective information. For example, as

Should I take AI?

Answer 1:
AI deals with many interesting topics, such as NLP, Vision and KR.
But it has many assignments which consist of writing papers.
You have little experience writing papers.
So it could be difficult.
I would not recommend it.

Answer 2:
AI has many programming assignments and it covers a lot of interesting
topics, such as NLP, Vision and KR.
So it should be quite interesting.
I would recommend it.

Figure 1-3: Example of output generated by ADVISOR II

will be discussed in the thesis, it is not possible to determine whether a class has *many assignments* uniquely based on the number of assignments that is stored in a knowledge base describing classes. As a consequence, in ADVISOR II, the content selection module (*i.e.*, the module determining what to say) not only extracts information from a knowledge base, but also *evaluates* it, to produce user and situation specific judgments. To perform this task, the evaluation system has access to a user model and a set of evaluation functions, in addition to the knowledge base.

Evaluation functions are inference rules which link observations to scalar judgments. For example, an evaluation function specifies that for a student who has no experience in programming, the presence of at least one programming assignment in a class triggers the evaluation of the class as high on the scale of programming. Both the knowledge-base and the user-model are expressed in CLASSIC [Resnick et al. 90] a KL-ONE-like knowledge representation language. Evaluation functions convert the information extracted from these sources into the Functional Description (FD) format used uniformly in the rest of the system. In addition, the evaluation system annotates the propositional content extracted from the knowledge base with pragmatic indications defining its argumentative function. In contrast, most other systems gather all the input to the surface realization component exclusively from an objective knowledge base, which prevents any form of sensitivity to the pragmatic situation.

The evaluation system then uses topoi to link individual judgments into argument chains. For example, a topos specifies that the higher a class on the scale of programming, the more time consuming it is. Topoi are domain-specific commonsense relations between scalar evaluations. An argument chain might specify that the more programming, the more time-consuming; the more time-consuming, the more difficult; and finally, the more difficult, the less the student should take the class. Another chain might specify that for a student interested in programming, the more programming, the more interesting (a user-specific topos); and the more interesting, the more the student should take the class. Topoi are encoded in a FUF grammar and chaining through the topoi base is performed through unification using the FUF system only. Such use of the FUF formalism for non-linguistic applications is unique to the ADVISOR II system and demonstrates the expressiveness and flexibility of the FUF formalism.

The output of the evaluation system is a set of argument chains. Each chain is rooted in an observation, which is extracted from the knowledge base, and assumptions about the user, which are extracted from the user-model. Both the observation and the user-assumptions form a conceptual network, which is the basic content that the generator eventually expresses. This conceptual network is annotated by a description of the argumentative intent of the answer to be generated.

This set of argument chains is then passed to the paragraph planner, which has responsibility for two tasks: (1) it selects an orientation for the final answer, evaluating the trade-offs between contradictory argument chains; and (2) it constructs a hierarchical paragraph plan which combines several argument chains into a coherent rhetorical structure. At this point the implemented paragraph planner does not make the decisions in the first subtask, but instead relies on user input or on random decisions to evaluate trade-offs. The output of the paragraph planner is a text plan, where each unit is either an argumentative evaluation or a conceptual network corresponding to an observation annotated by an argumentative evaluation. The paragraph planner is not a focus of this work.

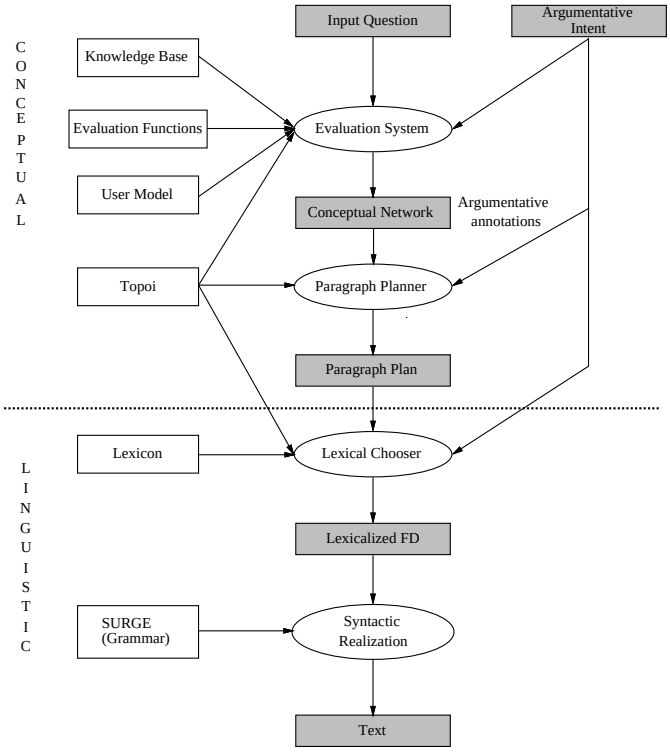


Figure 1-4: Architecture of ADVISOR II

The main focus of the work is on highlighting the impact of the pragmatic situation, and specifically, argumentative factors, on surface realization. As a consequence, the important aspect of the content selection and organization module described above is that the same knowledge source, topoi, is used both to perform content selection *and* to describe lexical entries in the lexicon. Topoi therefore form a bridge between conceptual and linguistic decisions.

The paragraph plan, which contains propositional content in the form of a conceptual network and which is annotated by argumentative features, is then processed by the lexical chooser. The lexical chooser determines how to segment the conceptual networks into clauses, the structure of each clause (clause planning), the structure of each NP (NP planning), and finally selects all open-class lexical items in the answer. The output of the lexical chooser is a fully lexicalized functional description for the whole paragraph. The lexical chooser is implemented as a FUF grammar. Three important features characterize the lexical chooser:

- Clause and NP planning is moved from the content planning to the lexical chooser. In previous systems, clause and NP planning were performed implicitly by the content selection module, often relying on hand-coded rules (for example, when using a phrasal lexicon [Kukich 83b; Hovy 88a; Jacobs

85]. In this work, both are performed by the lexical chooser, which considerably improves paraphrasing power.

- Lexical entries include indications of which argumentative function each word can have. Lexical selection is constrained by these indications and the argumentative intent the answer is to realize.
- The same argumentative intent can be realized by a variety of evaluative expression. The lexical chooser can handle such floating efficiently, using FUF's control mechanisms (bk-class and wait). Specifically, the lexical chooser can realize an argumentative evaluation at the following ranks: connective, main verb, adjective and determiner sequence.

The lexicalized FD is finally sent to the syntactic realization component, which is the SURGE grammar, processed by the FUF system. Closed-class lexical items (in particular the determiner sequence) are selected by SURGE.

The main focus of this work has been to develop a lexical chooser and a surface realization component which are sensitive to argumentative intent. The ADVISOR II system demonstrates that the specification of the argumentative intent can be derived from available knowledge sources using standard AI techniques, and that, therefore, the lexical chooser presented in this work can be effectively driven by a practical system.

1.4. Contributions

The main contributions of this work fall into three categories: definition of new lexical choice techniques, definition of an extended formalism for generation and application of the generation technology to a practical domain.

1.4.1. Lexical Chooser with Extended Coverage

The lexical chooser developed in this thesis is capable of generating the following constructions:

- Judgment determiners: *few, a few, little, a little, some, a lot, lots of, plenty of, several, many, much, a (good, great) many, most, a [adj] number of, a [adj] (quantity, amount) of.*
- Scalar adjectives with appropriate intensifiers, *e.g., interesting, hard...*
- Verbs with connotation, *e.g., require, enjoy...*
- Composite processes, *e.g., make difficult...*
- Non-predicative modifiers in a compositional manner, *e.g., a programming class, an AI topic...*
- Argumentative connectives: *so, therefore, because, since, but, although.*

These constructions were not constructed in a principled and compositional manner in previous generators.

1.4.2. Lexical Chooser with Extended Flexibility

In addition, the lexical chooser is capable of flexibility in building the linguistic structure from a conceptual description which is not committed to an a priori linguistic form:

- At the clause level, by selecting freely a perspective on a non-hierarchical input conceptual network. The lexical chooser can therefore select between:

  AI covers many interesting topics.
  Many AI topics should interest you.

- At the clause level, through the ability to merge two conceptual relations into a single linguistic clause, by either using a verb with connotation or a composite process. The lexical chooser can therefore select between:

I struggled with AI.
I have taken AI and I have found it difficult.

Programming makes AI time-consuming.
AI requires a lot of programming, therefore it is time-consuming.

- At the NP level, by dynamically mapping modifiers to either predicative pre-modifiers, non-predicative pre-modifiers, relative clauses or prepositional phrases. In contrast, in previous generators, the syntactic type of NP modification for a given modifier was either specified in the input to the generator, or determined statically, by specifying, for example, that color modifiers are to be always realized by a pre-modifiers. The lexical chooser can therefore select between:

  A programming assignment.
  An assignment which requires programming.

  An interesting topic about theory.
  A theoretical topic which should interest you.

- At the NP level, to realize countable sets by selectively expressing the cardinality of the set (expressing the number of elements in the set by using a cardinal or a judgment determiner), its extension (listing the elements of the set), its intension (the characteristic property identifying the set) and its reference (the domain out of which the set is defined). The lexical chooser can therefore select between:

  Three interesting topics.
  NLP, Knowledge representation and Vision.
  Many AI topics.
  Several interesting topics in AI - NLP, Knowledge representation and Vision.
  Three topics which are covered in AI and which should interest you.

This flexibility allows for structural paraphrasing and significantly increases the variety of the generated output and its fluency.

Finally, the lexical chooser is sensitive to several aspects of the pragmatic situation, characterized by the argumentative intent of the speaker, politeness criteria, style constraints, focus and thematic development.

1.4.3. SURGE: a Large Portable Reusable Syntactic Realization Grammar

SURGE is a large portable reusable syntactic realization grammar for English.[4] It processes the specifications produced by the lexical chooser and enforces syntactic constraints such as agreements, ordering constraints, choice of closed-class lexical items (determiners, prepositions) and morphological inflections. SURGE is based on principles of systemic grammars, as described in [Halliday 85] but it also integrates some ideas derived from HPSG [Pollard & Sag 87].

SURGE takes as input a lexicalized specification, where all open-class lexical items are specified and where the function of each lexical item within the clause is specified in semantic terms (*e.g.*, using functions such as *agent* or *attribute*). It produces as output a complete syntactic specification, which is then linearized to produce an English sentence.

SURGE has been used in many research projects in generation (five different projects at Columbia, and I am aware of five currently active projects outside of Columbia). FUF and SURGE have been distributed to more than 30 research sites and are being evaluated actively by a large number of researchers. This exposure has ensured the portability and reusability of SURGE. It has also validated the FUF environment as a viable programming environment making generation a more accessible technology.

[4]SURGE stands for Semantic Unification-based Realization Grammar of English.

1.4.4. FUF: an Extended Formalism for Text Generation Systems Development

FUF is an implementation of the FUG formalism which includes several extensions making the system more expressive and more usable.[5] The FUG formalism has already been recognized as a useful way of implementing syntactic realization grammars for generation. It was used in two of the first and most influential generation systems - TEXT [McKeown 85] and KAMP [Appelt 85].

FUF extends on this early work in several ways:

- FUF is significantly more efficient and supports much larger grammars than earlier FUG systems.
- FUF is used to implement more tasks within the generation process in a uniform manner. Beyond syntactic realization, FUF is used for lexical choice and part of content determination.
- FUF includes a novel typing system and supports inheritance, increasing the expressiveness and the conciseness of the formalism.
- FUF includes several innovative control mechanisms allowing the implementation of larger and more complex grammars while retaining an acceptable processing time.
- FUF includes mechanisms allowing the modular specification of grammars and their interaction with external knowledge sources, thus allowing the development of larger and more robust and readable grammars.

As mentioned above, FUF also offers significant advantages over competing generation formalisms (NIGEL and MUMBLE) because it allows the specification of non-deterministic, declarative and bi-directional grammars. These features allow it to be useful for dealing with tasks computationally more complex than just syntactic realization.

1.4.5. Advisor II: a Prototype of an Argumentative Generation System

The final contribution of this work has been the implementation of a complete generation system, integrating all the elements touched upon in the previous paragraphs into a coherent practical application. The ADVISOR II system has served as a testbed demonstrating the usefulness of the techniques discussed in this dissertation.

ADVISOR II is a question-answering system which provides university students with advice about which courses to take in a semester. It is best described as an argumentative system whose main task is to convince a user to either take or avoid a given class. The generation of evaluative expressions is, therefore, critical to the performance of this system.

ADVISOR II uses the lexical chooser described in this work and the SURGE grammar for the realization component. Its content determination module produces the features required by the lexical chooser to control the selection of most of the linguistic constructions discussed in this work.

1.5. Guide to the Rest of the Thesis

The rest of the thesis is organized as follows: in Chap.2 the role of argumentation in generation is presented. The theory of argumentation used in this work is discussed in more detail and the role of topoi as a bridge between content determination and surface realization is highlighted. This chapter provides most of the motivation for the work on lexical choice presented later.

The following three chapters can be read separately. They describe the portable and reusable components developed in this thesis. A reader interested in using the FUF environment and/or the SURGE surface generator can use these three chapters as a description of the system. Specifically, Chap.3 presents the base FUG formalism and compares it

[5] FUF stands for Functional Unification Formalism.

to existing generation formalisms. While this chapter does not provide new results, it is, to my knowledge, the more detailed description of the FUG formalism available to date.

Chapter 4 motivates and introduces the extensions that I have added to the FUG formalism when developing the FUF system. Lexical choice when dealing with floating constraints like argumentation required new facilities. Extensions include different forms of typing, control mechanisms to make unification more efficient, and tools to improve the modularity of large grammars written in FUF.

Chapter 5 presents the SURGE syntactic realization grammar. This grammar expects an input where all open-class items are specified. It is portable and domain-independent. The form of the input expected by the grammar is motivated and presented in detail and the main linguistic issues underlying its design are discussed. In particular, I have identified places in the grammar where evaluative expressions can be realized and I have specifically developed the grammar for determiner sequences which is often used to express evaluations.

Chapter 6 describes the lexical chooser developed in this work. The chapter describes which conceptual material can be mapped to each linguistic category (clause, NP, adjective, determiner sequence and connective). An analysis of certain semantic properties of each linguistic category is used to determine which information must be present in the input to allow for the generation of a variety of linguistic structures. This analysis imposes constraints on the form of the input to the lexical chooser. Finally, for each category, the mapping from conceptual material to lexical decisions is described.

Chapter 7 describes the implementation of ADVISOR II. Finally, the conclusion summarizes the contributions of this work, highlights its limitations and outlines future work.

Chapter 2
Using Argumentation in Text Generation

The motivation of this research is to improve fluency in text generation and make generated text more sensitive to the pragmatic situation. The assumption underlying this work is that these goals can be achieved by making lexical choice more flexible and considering the speaker's communicative goals at all levels of the generation process. To illustrate this strategy, I have focused on a particular class of communicative goals - expressing the argumentative intent of the speaker - and on the various methods by which this goal can be linguistically realized. An argumentative intent of the speaker corresponds to the goal of convincing the hearer of a certain conclusion. The main linguistic devices used to fulfill this goal are evaluative expressions. I explain in this chapter how to make a text generator sensitive to the speaker's argumentative intent and capable of generating a wide variety of linguistic forms (different evaluative expressions) to fulfill this intent.

The method presented in this dissertation emphasizes that the communicative goal affects decisions at all levels of the generation process. For example, to convince a hearer of a conclusion, the speaker can (1) select appropriate facts supporting the conclusion, (2) organize the information in a convincing manner and (3) choose a phrasing that reflects the orientation of the speaker. That is, using technical terms to refer to each of the stages of the generation process, the speaker's argumentative intent influences content determination, content organization and surface realization.

The questions addressed specifically in this chapter are: how is a complete generation system made sensitive to an input communicative goal; and specifically, what are the different roles the argumentative intent plays in the generation process. To make this problem concrete, I consider how argumentation influences the implementation and architecture of the ADVISOR II system. ADVISOR II is a question-answering system providing advice to university students on how to select courses. When generating an answer, the system has a goal to convince the user to either take or avoid a specific class.

This chapter focuses on the role of argumentation in the design of this generation system. I first briefly describe the architecture of a complete generation system. I then review work in linguistics which uses argumentation as a central element of a semantic theory, specifically the theory of argumentation in language developed in [Anscombre & Ducrot 83]. From this theory, I select the notion of topoi, argumentative rules explaining the connection between a pair of evaluations, as central to generation. The role topoi can play at all levels of the generation system is then explained. The main claim of the chapter is that topoi can serve as a bridge between the decisions made at the content planning stage and at the surface realization stage, and can, therefore, play a central role in making generation more sensitive to the pragmatic situation.

2.1. A Characterization of Generation Problems

Surface generation is best characterized as the process of making choices between alternate linguistic realizations under the constraints specified in the input to a text generator. Depending on the practical application, the input can take different forms - streams of numbers in report generation, traces of rule activations in expert system explanation, knowledge-based descriptions in data-base interfaces or in text translation. In a strong sense, the application determines what knowledge is available to drive the generation process. In another direction, however, the analysis of quasi-paraphrases of a linguistic constituent can also determine what input must be present to explain the distinction between alternate linguistic realizations. In that sense, the linguistic system determines what knowledge must be made available to drive the generation process.

The problem faced by the designer of a text generation system is to match the information provided by an application with the requests of a linguistic component. In this chapter, it is shown how the linguistic theory of argumentation of Ducrot and Anscombres can help to bridge this gap, in identifying a middle ground between the knowledge necessary to distinguish a whole class of linguistic phenomena and the type of knowledge used by applications [Anscombre & Ducrot 83]. This bridge takes the form of an abstract descriptive tool - the topos. Topoi are gradual inference rules that capture ''common sense'' relations. For example, *the more an activity is interesting, the more people want to do it*. Topoi are used in D&A's theory of argumentation to explain the semantics of connectives like *but* or of argumentative operators like *however* or *at least* [Anscombre & Ducrot 83]. Recently, topoi have also been used in the description of lexical items like verbs and adjectives [Bruxelles *et al* 89; Bruxelles & Raccah 91]. Topoi are also sufficiently abstract to serve as a convenient bridge between the type of knowledge representation used in knowledge-based applications and a linguistically motivated text generator.

Thus the main advantages of using the theory of argumentation in language for text generation can be summarized by two points:

- argumentation accounts for linguistic choices like gradual adjectives, contrastive connectives or vague quantifiers. These phenomena pose problems to most semantic theories, and are often handled in an *ad-hoc* manner by generation systems.
- argumentation establishes a bridge between the knowledge provided by an application and the needs of surface realization.

The FUF text generator system as used in the ADVISOR II system illustrates how the handling of argumentative features throughout the generation process, from content determination to surface realization, helps control linguistic decisions that were not considered in previous work in text generation. I show how FUF uses argumentative constraints to perform the following tasks:

- Content determination
- Content organization
- Lexical choice

The main points of the chapter are that argumentation is necessary to perform a wide range of lexical choice operations and that the argumentative information needed to perform these operations can be derived naturally from the generation process prior to lexicalization by using a single conceptual tool, the topoi, at *all levels* of the generation process.

The rest of the chapter evolves as follows: first, the different subtasks that a text generator must fulfill are enumerated, and after briefly reviewing the theory of argumentation, I discuss how argumentation can be combined with and complement existing generation techniques for each subtask up to lexicalization. Finally, I explain how the argumentative information included in the input to the lexical choice module is used to select classes of words that were not considered in generation prior to this work.

2.2. The Domain Problem: Generating Advice-Giving Paragraphs

Consider the task of advising a university student about the courses he should follow. A system providing this type of advice has to interact with a user and provide appropriate evaluations of the courses about which the student may inquire. A typical interaction between a student and a real academic advisor is shown in Fig.2-1.[6]

I have developed an explanation component for a system called ADVISOR II, which models this task. ADVISOR II is derived from an earlier system developed in the same domain at Columbia from 1982-1987 [McKeown *et al* 85; McKeown 88]. The focus of ADVISOR II is on one of the subtasks of such an advising system: how to provide an

[6]This example is an excerpt from transcripts of advising sessions between human academic advisors and students recorded at Columbia University in 1984.

Student - How is PLT I? Somebody told me it's going to be a lot of work and it's pretty hard.

Advisor - I can't say how hard it is. There is going to be some work, and it requires quite a lot of programming.

Student - But isn't that really recommended to take it for graduation or...?

Advisor - Well, I think it falls into the same category as AI and Operating Systems. I mean, those are like the same level of courses and all three are very, very important courses.

Figure 2-1: Fragment of a naturally occurring advising session

evaluation of a course and communicate it to the hearer. The paragraph shown in Fig.2-2 illustrates the capabilities of the system. The answer is tailored to a particular student, assuming that the student has expressed interest in Natural Language Processing (NLP) and programming. Note that the answer uses vague determiners (*most* and *many*) and a scalar adjective (*interesting*). These lexical decisions contribute to the natural and fluent character of the answer.

Should I take AI?

AI covers many topics related to NLP,
and most of the assignments involve programming.
It should be an interesting class.

Figure 2-2: A paragraph generated by ADVISOR

In order to produce such a paragraph, many operations need to be performed: relevant information about the course and the student must be gathered, an evaluation of the course for the particular student must be computed, the information supporting that evaluation must be organized and finally the evaluation must be converted into words and linguistic constructions. One view of the generator architecture is shown in Fig.2-3.[7] In this figure, active processes are shown in rectangle boxes and knowledge sources in shaded boxes. Edges indicate data flow. The figure highlights the central role of topoi in the system, as a bridge between conceptual and linguistic decisions.

These modules perform the following tasks:

- **Goal and content determination**: The system reasons about the interaction situation and determines what goals must be achieved by the answer. In the advising domain, the goal can be to inform the student, to recommend taking a course or against taking it. In ADVISOR II the goal of the answer is given as an input to the system. This goal is the argumentative orientation of the answer. To support this goal, the system extracts relevant information from a knowledge base. In the example, the system decides to include the fact that AI covers a certain set of topics and requires a certain type of assignments. Information on the topics and assignments of classes is stored in the knowledge base. The system must also perform some inferencing to conclude, for example, that NLP-related topics and programming assignments make up for an interesting class. The output of this stage is a collection of conceptual propositions, in a predicate-argument notation. In addition, in ADVISOR II, all propositions are annotated by their argumentative function.

[7] When comparing this figure with Fig.1-4, the perspective taken here is to highlight the central role of topoi at the articulation between conceptual and linguistic decisions. In the ADVISOR II implementation, goal and content determination are performed by the evaluation system and content planning is performed by the paragraph planner.

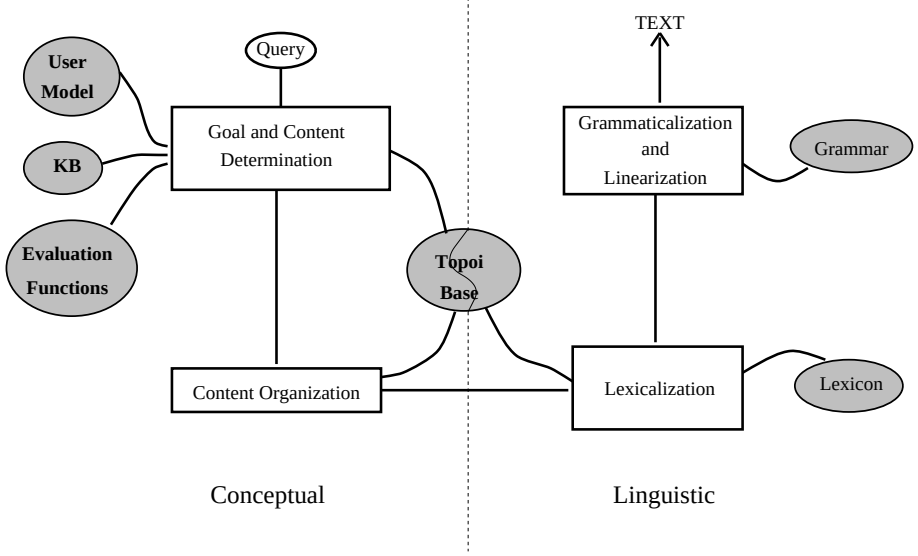


Figure 2-3: Architecture of a text generator

- **Content organization**: The system structures the information extracted at the previous stage into a network of rhetorical and/or logical relations. For example, the paragraph in Fig.2-2 contains three propositions structured as `(premise conclusion)`, where `premise` itself is a conjunction of two propositions. The information extracted from the knowledge base is also filtered at this stage: for example, the answer does not mention the fact that AI covers many mathematical topics. This is also the time when an ordering is imposed on the information to be conveyed in the answer.
- **Lexical choice**: The conceptual structures produced by the content determination and organization modules are mapped onto linguistic structures, with heads and arguments. For example, the system chooses among structures like *AI assignments involve programming*, *programming assignments in the AI domain* (a noun phrase) and *AI involves programming assignments*. In addition, lexical items (words) are selected. For example, the system chooses between *assignments involve programming* and *assignments require programming*, or between *assignments* and *assignments*. The system also decides which determiners to use, *many* or *most* or unmarked ones like *the*. The output of this stage is a network of grammatical relations with lexical items appearing at each node.
- **Grammaticalization and linearization**: The system determines paragraph and sentence boundaries, the syntactic structure of the propositions (subordination and embedding), pronominalization of selected references and scoping relations between arguments, grammatical tense and modality, agreement and ordering constraints. The syntactic structure is linearized into a string of words. Morphology is handled at this stage.

To perform these tasks, the system accesses various knowledge sources. A knowledge base written in CLASSIC, a KL-ONE like knowledge representation system, describes courses offered in the semester [Resnick et al. 90]. The user model is also described in CLASSIC and contains information such as which courses the student has already taken, what are his interests and experience. Specific to the approach in this thesis are a set of topoi and a set of evaluation functions which are used to map the content of the knowledge base to situation specific utterances. These are described in detail in Sect.2.4.1. Finally, the linguistic components use a lexicon and a grammar described in the FUF formalism [Elhadad 91a]. The lexicon is described in Sect.2.6.

In the rest of the chapter, I focus on the task of expressing an evaluation of a class: the linguistic theory of argumentation of Ducrot and Anscombres is first briefly presented. I then present a new model of evaluation, distinguishing between three types of evaluations and highlighting the role of topoi in the definition of the most complex type. I then describe how topoi are used to perform content determination and organization, and lexicalization, emphasizing their role as a bridge between conceptual and linguistic decisions.

2.3. Previous Work: A Brief Review of the Theory of Argumentation

This work makes heavy use of the notion of topoi, introduced in the linguistic theory of argumentation of Ducrot and Anscombres [Anscombre & Ducrot 83]. It also uses a theory of lexical semantics where quasi-synonyms differ by their connotation. I first review Anscombres and Ducrot's theory of argumentation [Anscombre & Ducrot 83] and then discuss how Osgood's work on scalar lexical semantics can be related to this theory [Osgood, Suci and Tannenbaum 57].

2.3.1. Argumentation in Language

The theory of argumentation in language itself is introduced in [Anscombre & Ducrot 83]. I also rely on developments presented in [Raccah 87; Bruxelles & Raccah 91]. The goal of this section is primarily to introduce terminology that is used in the rest of the thesis.

The linguistic theory of argumentation relies first on a distinction between *sentence* and *utterance*: a sentence is an abstract linguistic object while an utterance is the concrete event occurring when a speaker says a sentence in a certain situation. The theory of argumentation is concerned with sentences and not utterances. The point of the theory is that the semantics of a sentence includes indications about how it can be presented, in all situations, to support a set of conclusions, and that, therefore, argumentation is not a completely pragmatic phenomenon, but also concerns the semantics of a language. *Argumentative orientation* is by definition the specification in the semantics of a sentence of which conclusions can be supported by its enunciation. The main observation of the theory is thus that there is argumentation within language.

In addition, the theory of argumentation assumes that the argumentative orientation of a sentence can be described in a restricted formalism, based on semantic rules called *topoi*.[8] Topoi are gradual inference rules of the form: *the more/less X is P, the more/less Y is Q*, where *X* and *Y* are semantic elements present in the sentence, and *P* and *Q* are called *topic scales*. For example, *the more a class is difficult, the less a student wants to take it* is a topos used in the ADVISOR II domain. I view a topos as the combination of four primitive relations:

- An evaluation of the entity *X* on the scale *P* (*e.g.*, the class on the scale of difficulty).
- An evaluation of the entity *Y* on the scale *Q* (*e.g.*, the student on the scale *desire to take the class*).
- The expression of a gradual relation between *P* and *Q* (*e.g.*, the harder the class, the less desire).
- The expression of a topical relation between *X* and *Y* (*e.g.*, the class on the left side is the class rejected on the right hand side).

The argumentative orientation of a sentence does not determine completely which topoi can be triggered by its enunciation. It only constrains some aspects of the description of a topos that can felicitously be applied to the utterance in a given situation. Consider, for example, the following sequences:

- *AI is hard. You should not take it.*
- *AI is hard. You should take it.*
- *AI is hard. You would enjoy the challenge.*

The first sentence sounds natural while the second one sounds at first surprising. One might be inclined to conclude that the clause *AI is hard*, therefore, triggers the activation of a topos of the form *the more a class is difficult, the less a student wants to take it.* This is not, however, a valid conclusion, because even if the second sentence sounds surprising, one might imagine situations where it makes sense. For example, situations where the student has expressed a taste for challenge, as illustrated by the third example; that is, situations where the topos *the more something is difficult, the more person X wants to do it.* While such situations may be rare, the fact is that the topoi that are triggered by the single sentence *AI is hard* are not completely determined by its linguistic form. When more linguistic markers are available in a sentence, however, more constraints can be derived and the set of topoi which

[8] Topos singular. Topoi plural.

can be triggered becomes smaller. For example, if instead of considering just the first clause *AI is hard*, one considers the whole sequence *AI is hard, you should not take it* then only the topos *the more a course is difficult, the less a student wants to take it* (formally, `/+ Difficult(Action), - Want(Agent(Action), Do(Action))/`) can be used in all situations.

As for the single clause *AI is hard*, it can be assumed that any topos triggered by this clause must include an evaluation of an entity of type *class* on the scale of difficulty. That is, of the four primitive relations that constitute a topos, only one is constrained by the linguistic form of the clause. In general, the theory predicts that lexical items and syntactic constructions constrain which topoi can be triggered from the utterance of a sentence in a given situation. For example, the constraint above would be created by the selection of the adjective *hard* and its use in a predicative construction. Other classes of words impose different types of constraints: for example, connectives like *but* or *because* impose constraints not on which scales are activated but on the orientation of the evaluation on the scales [Ducrot *et al* 80].

The claim of the theory of argumentation in language is that the set of conclusions that can be supported from the enunciation of the sentence *AI is hard* is constrained by the semantics of its components - namely the noun *AI*, the adjective *hard* and the predicative relation between them. From a generation perspective, the same claim can be phrased as: the argumentative orientation of a sentence partially determines what lexical items and syntactic relations can be used in the sentence.

I close that review by highlighting the function of topoi in the theory: on one hand, topoi are part of the domain knowledge of speakers, on the other hand, topoi-related features are also included in the lexicon and in the grammar, for each lexical item and syntactic construction. In that sense, and that is the main claim in this chapter, topoi can serve as a bridge between the conceptual representations used in an application and the linguistic demands of a text generator.

2.3.2. Scalar Lexical Semantics and Scalar Connotations

One of the claims of the linguistic theory of argumentation is that the semantics of certain lexical items is scalar. From a different perspective and with different goals, Osgood has also developed a theory of lexical semantics relying on the notion of scalar dimensions to meaning.

Consider, for example, the verbs *require, necessitate* and *demand* in a context like *AI requires many assignments*. All three verbs have a scalar connotation that projects an evaluation of their subject on the scale of difficulty. In contrast, the verbs *have, include* and *involve* lack this connotation. The notion of connotation that I use here is extensively reviewed in [Kerbrat-orecchioni 77]. The underlying intuition is, when considering a group of words like *{require, necessitate, demand, have, include}*, to distinguish between a ‘‘core meaning’’ which is shared by all elements, and ‘‘connotations’’ which capture the ‘‘added meaning’’ of each element. There is no logical relation between the core meaning of a word and its connotations. For example, the evaluation of AI as a difficult class when the verb *require* is used is independent of the denotation of *require* (which is the conceptual relation `assignments-of`) and not motivated by it (the relation `assignments-of` does not imply the evaluation of the class on the scale of difficulty).

One family of connotations has been studied under the term of ‘‘relations of fusion’’ in [Gross 75]. The linguistic test formulated by Gross is (quoted in [Danlos 87a, p.46]):

- *Bob gobbled his egg. = Bob ate his egg by gobbling it.*
- *Bob devoured his pizza. = Bob ate his pizza by devouring it.*

These equivalences indicate that there is a relation of fusion between *eat* and *gobble* and between *eat* and *devour*. In contrast, there is no such relation of fusion between *eat* and *smoke* for example:

- ** Bob ate his soup by smoking it.*
- ** Bob smoked his cigarette by eating it.*

A similar test can be devised for testing another type of connotative relation between verbs. For example, the following equivalences indicate that *enjoy* (a class) is a connotative variant of *take* (a class):

- *Bob enjoyed the AI class == Bob took the AI class and enjoyed it*
- *Bob struggled with the AI class == Bob took the AI class and struggled with it.*

Scalar connotations are the particular type of connotations that convey a scalar evaluation. In [Osgood, Suci and Tannenbaum 57], a technique to recognize scalar connotations is described. The technique, known as the *semantic differential* relies on a statistical analysis of the answers of subjects to a test associating a lexical item with a set of bipolar scales.

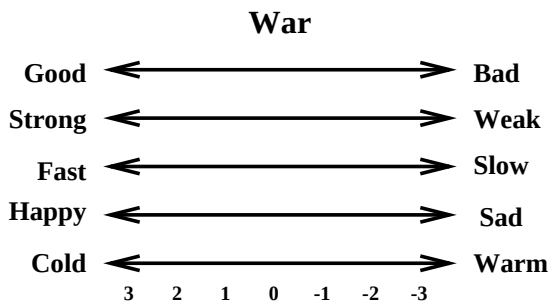


Figure 2-4: Osgood's semantic differential

One of the goals of Osgood's study was to identify the scales playing a role in explaining the ''conceptual distance'' between words, by using factor analysis to identify independent scales. The experiment started by identifying 50 scales by asking subjects to associate adjectives with a random sample of nouns. (In another version of the experiment, some 289 adjective pairs were identified from an analysis of Roget's thesaurus.) Then 100 subjects were asked to rank 20 nouns on these 50 scales, with a ranking from -3 to 3. A factor analysis of the results identified three major dimensions along which judgments were made: evaluative (good-bad), potency (strong-weak) and activity (fast-slow). A traditional criticism of the method is that the original set of scales is quite arbitrary and can, therefore, taint the final results. An example of such a test is shown in Fig.2-4. The results of such an analysis provide a sort of ''connotative profile'' for lexical items.

In this case, I am interested in choosing lexical items which can be used to express a scalar evaluation. Lexical items carrying a scalar connotation are prime candidates for this task. For these purposes, however, one does not need to uncover a set of ''universal scales'' that could define the independent dimensions of some conceptual space. In contrast, one only needs to identify the connotations of items on the domain-dependent argumentative scales that have been identified in the domain. So to uncover the argumentative connotations in the domain, a much simplified form of the semantic differential could be used, presenting grids of evaluation similar to the one in Fig.2-4, but only with the seven scales of the domain presented in Sect.2.4.3 (applying the technique as advocated by its inventors themselves [Osgood, Suci and Tannenbaum 57, p.76]).

Another difference of the use of the technique made in this work from the original Osgood test is that partial syntactic constructions are tested as opposed to simple nouns. For example, the test question is: *when ''X requires Y'', X is <scalar evaluations> and Y is <scalar evaluations>*. This notion of construction is related to the notion of ''lexically open idioms'' of [Fillmore, Kay & O'Connor 88]. The underlying principle is that argumentation is not only conveyed by single words but by constructions that are conventionally associated with some pragmatic effect.

In the ADVISOR II system, scalar connotations are stored in the lexicon with each entry as discussed in Sect.2.6. Scalar connotations are expressed in terms of topoi, thus an important part of the lexical semantics in the system is expressed in argumentative terms. In our implementation, the connotations correspond to my intuition and have not been validated by a statistical test. The mechanisms for using the connotation information, however, would not be changed even if the intuition is not corroborated by statistical evidence. This technique only relies on the assumption that some statistically significant scalar connotations can be identified.

2.4. The Role of Topoi in Content Determination

In the ADVISOR II domain, consider a student asking the system: *Should I take AI this semester?* The system must first determine what the goal of the answer will be. ADVISOR II takes as input one of the following goals:

- Recommend the course to the student
- Recommend against taking the course
- Inform the student about some properties of the course.

In this chapter, I focus on the first two goals, which are argumentative in nature. The task of content determination is, given this argumentative intent as input, to gather relevant information for inclusion in the answer. The system has access to several sources of information: a knowledge base describing the courses offered in the university, information about the student, including which courses he or she has already taken, topics of interest, and experience. The eventual paragraph produced by ADVISOR II as shown for example in Fig.2-2 presents only a subset of the information gathered at this stage. Content determination produces the larger set of propositions that are relevant to the evaluation of the class for a specific student.

The focus of this work is to study how evaluations can be expressed in language. In this section, I first present a model of what constitutes an evaluation in terms of topoi. I then show how the system produces evaluations by extracting information from the knowledge base and relating it to the argumentative intent of the answer it produces. The next section discusses how these evaluations are mapped onto linguistic structures.

2.4.1. Three Types of Argumentative Evaluation

Topoi have been introduced as simple gradual rules, and their definition informally written as in */+assignments, +difficult/*. Such notation is an abbreviation for a more complex structure. Consider the topos: */the more a class has assignments, the more difficult the class/*. By definition, this topos is the composition of two gradual evaluations: on the left-hand side, an evaluation of the cardinal of the set of assignments, on the right-hand side, an evaluation of the class on the topical scale of difficulty.

An evaluation is an answer to a question of the form *how P is X?*. For example, the left-hand side evaluation is an answer to the question *how many assignments are there in AI?*. I distinguish between three possible types of answers to this question. Following [Bruxelles & Raccah 91], I distinguish between *simple* and *composite* topical evaluations. I also introduce a distinction between *simple absolute* and *simple relative* evaluations.

The first type of answer to the *how many* question is an exact answer: *AI has six assignments* which corresponds to a *simple absolute evaluation*. It consists in ‘‘measuring’’ an entity on a conceptual scale. For example, one can measure the number of assignments given in a class, or the number of topics a class covers. In practical terms, such measuring can be performed by looking up the knowledge base of the system. Using the same notation as [Bruxelles & Raccah 91], I note the result of such a measuring operation as: `<CONCEPTUAL-SCALE, VALUE>`. For example: `<cardinal({hw, assignments-of(AI, hw)}), 6>` stands for the simple absolute evaluation that AI has six assignments.

When the scale on which the evaluation is performed is not ‘‘measurable’’ in the knowledge base, then the value of a simple absolute evaluation is by convention either `+` or `-`. For example, the difficulty of a class is not a primitive notion in the knowledge base: it is not very meaningful to characterize a class as ‘‘difficulty level 5’’. Instead, this evaluation must be derived by an argumentative inference in a specific situation (taking into account the goal of the speaker and information about the student). In such a case, the evaluation is noted as `<difficulty(AI), +>`. Since such evaluations cannot be simply ‘‘measured’’, they can be derived in two possible ways: they are either derived by argumentative inference or they correspond to an *a priori* bias of the answer. I discuss below how argumentative inference is modeled in the system.

Another type of answer to the question *how many assignments are there in AI* is to compare the number of AI assignments with some other value. I call such evaluations, not covered in the model of [Bruxelles & Raccah 91], *simple relative evaluations*, and note them as in: `< cardinal({hw, assignments-of(AI, hw)}), +`

`{comparison-class} >`. This formula can be read as: the set of assignments in AI is large in relation to some comparison class. The comparison class is a pragmatic variable that can be instantiated in a given situation of enunciation, and can be constrained by certain elements in the sentence. For example, the following reference sets can be used in different situations:

AI has more assignments than

- *Introduction to programming*
- *any other course in the University*
- *a typical class in computer science*
- *a typical class in the university*

The notion of comparison class is very general and has been used to provide a semantics to scalar adjectives (a comprehensive treatment of this notion is provided in [Klein 80]), to account for comparatives (cf. [Rayner and Banks 90] for a practical application of Klein's model) and for constructions like *big for an elephant* [Ludlow 89].

The third type of answer is given by evaluating the scale in terms of another evaluation. For example, the number of assignments would be qualified as being a number such that AI is difficult. Such *composite evaluations* measure a conceptual scale by using another evaluation. They are of the form: `< conceptual-scale, <conceptual-scale, value> >`. For example:

```
< cardinal({hw, assignments-of(AI, hw)}),
                    + <difficult(AI), +> >
```

This recursive definition captures the intuition that an evaluation is a way of looking at an entity. I evaluate the number of assignments in a class as it relates to the difficulty of the class; that is, as a criterion to conclude how difficult AI is.

The notion of composite evaluation is useful in distinguishing between *linear* and *non-linear* topical scales (a terminology used in [Klein 80]). A linear scale is a scale for which only one criterion can be used to compute an evaluation. In contrast, several criteria can be used to evaluate on a non-linear scale. For example, the scale of difficulty is non-linear: a class can be difficult because it is theoretical, or because it has many assignments... In contrast, the scale of cardinality for sets is linear.[9]

The three different types of evaluations are depicted in Fig.2-5. In this figure, conceptual relations extracted from the knowledge base are shown with straight lines, and argumentative evaluations are shown with wavy lines, labeled with the scale. Sets are represented in circles, and individuals as simple labels. The figure depicts three topical evaluations that could be phrased respectively:

AI has six assignments.
AI has many assignments [more assignments than other classes].
AI requires many assignments [so it is difficult].

A composite evaluation of the form `<c1, <c2, v>>` corresponds to a canonical topos, of the form */the more c1, the more c2/*. The right-hand side of Fig.2-5 can be seen as the depiction of the topos */the more a class has assignments, the more it is difficult/*. In this figure, the four relations that make up a topos (as discussed in Sect.2.3) appear as four arcs: the evaluation of the entity `HWS` on the scale of cardinal is shown with the rightmost wavy line, the evaluation of the entity `AI` on the scale of difficulty with the leftmost wavy line, the gradual relation between these two evaluations with the bottom implicative arrow, and the topical relation between the entities `AI` and `HWS` is shown with the knowledge-base `assignments-of` relation at the top.

[9]Non-linear scales induce only a partial order on the objects that are evaluated, which relates to the notion of partially ordered set used in [Hirschberg 85] in her study of scalar implicatures.

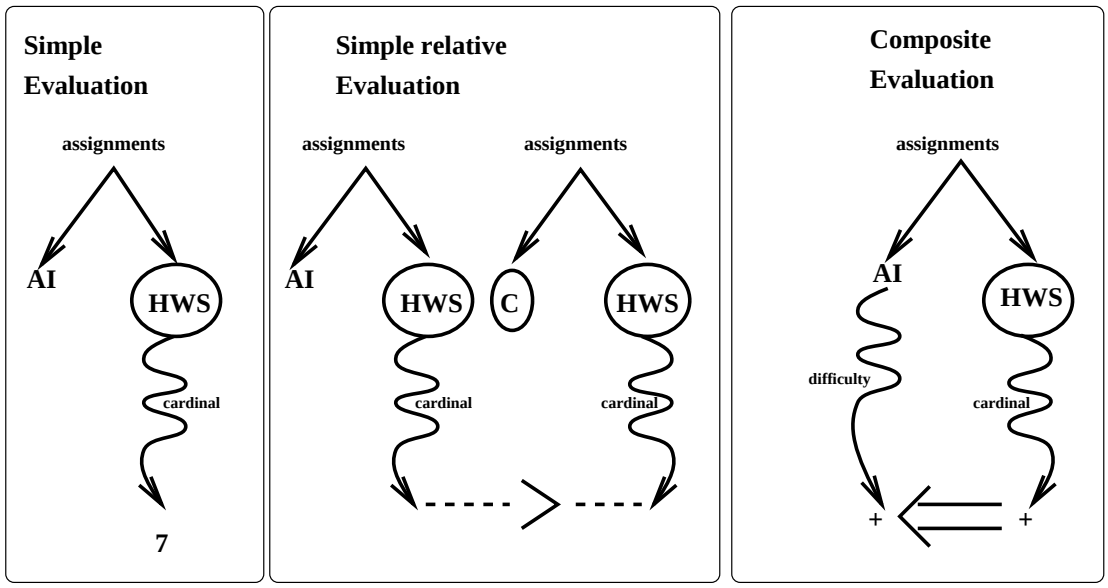


Figure 2-5: Three types of evaluations

2.4.2. Evaluation Functions: Producing Evaluations from Observations

The role of the content determination module is to produce a set of such argumentative evaluations that relate to the ultimate argumentative intent of the answer. This intent is given as input to the content determination module. The output is a set of propositions extracted from the system's knowledge base and a set of evaluations. While evaluations can be naturally related to the desired argumentative intent of the answer (which is itself an evaluation), there is no direct way to connect the knowledge base information available to the system to this same input. There is indeed a gap between these two types of propositions: evaluations are situation dependent (they depend on the student model and eventually on the argumentative intent of the answer) while knowledge base facts are not. This gap needs to be bridged to connect the information in the knowledge base to the argumentative intent of the answer given as input to the text generation system. The task of the content determination module is, therefore, both to relate evaluations to facts and to select only the set of facts that are appropriate for a certain conclusion.

From the generation point of view, the content determination method presented here addresses the ''expressibility problem'' noted in [Meteer 90]: evaluations are needed to produce linguistic elements like vague determiners or scalar adjectives, but the knowledge base only contains straight facts, with no evaluative information. There is, therefore, a need to introduce an evaluation procedure before sending the content to the linguistic realization component. I now explain how such evaluations are built in the ADVISOR II system. More detail is also provided on the knowledge sources required to perform this task: knowledge base, user model, scales and topoi base. I also discuss how the set of scales used in the domain is derived from a corpus-based analysis.

The argumentative intent is represented as an evaluation of the form `<take(student, AI), +>`. It is given as input to the system, and can, therefore, be considered as an *a priori* bias of the answer. The output of the content determination module which eventually leads to the production of the example in Fig.2-2 is shown in Fig.2-6.

The figure shows the four argumentative derivations that link information extracted from the knowledge base and the user model to the eventual evaluation `+/-take(student,AI)`. Each derivation is made up of three types of information:

1. A set of knowledge-base propositions called *observations*. An observation only contains objective facts about the classes and assumptions about the user.

```
QUESTION: Should I take AI?

(1) KB Observations:    assignments-of(ai, assignments1)
                        member-of(assignments1, a1),
                        activity-of(a1, paper-writing),
                        user-profile(experience, paper-writing, -)
Judgment:               <workload(AI), +>
Argumentative Linking:  /+ workload(AI), + difficult(AI)/
                        /+ difficult(AI), - take(student, AI)/

(2) KB Observations:    topics-of(ai, topics1),
                        member-of(topics1, nlp),
                        user-profile(interest, nlp, +)
Judgment:               <interest(AI), +>
Argumentative Linking:  /+ interest(AI), + take(student, AI)/

(3) KB Observations:    topics-of(ai, topics1),
                        member-of(topics1, logic),
                        area(logic, theory),
                        user-profile(experience, theory, -)
Judgment:               <theory(AI), +>
Argumentative Linking:  /+ theory(AI), + difficult(AI)/
                        /+ difficult(AI), - take(student, AI)/

(4) KB Observations:    assignments-of(ai, assignments1),
                        subset-of(assignments1, prog-assignments,
                                        activity(X, programming)),
                        cardinal(prog-assignments) > 2,
                        user-profile(interest, programming, +) ]
Judgment:               <programming(AI), +>
Argumentative Linking:  /+ programming(AI), + interest(AI)/
                        /+ interest(AI), + take(student, AI)/
```

Figure 2-6: Output of the content determination module

2. An evaluation supported by the observations called a *judgment*. A judgment is a simple absolute evaluation on a non-linear scale, using the terminology just introduced. A judgment can be seen as the left part of a topos anchored in the knowledge base.

3. Finally, each judgment is linked to the argumentative intent of the answer by chaining through topoi. Topoi are used here in a way very similar to the production rules of an expert system.

Thus the content generated contains two distinct types of propositions:

1. knowledge base facts, *e.g.*, `topics-of(AI, topics1)`.

2. scalar evaluations, *e.g.*, `<interest(AI), +>`.

An observation in the knowledge base is linked to the argumentative intent in two steps: a first evaluation is derived from the observation, then this evaluation is linked through a topoi-chain to the argumentative intent. The first step is performed by *evaluation functions*, which bridge the gap between knowledge-base information and evaluations. Evaluation functions map a collection of facts stored in the knowledge base to the activation of a scale. An example of evaluation function is shown in Fig.2-7.

Evaluation functions have the form of a query/conclusion pattern. The rule in Fig.2-7 can be read as follows: if there is in the knowledge base a set of topics `ts` such that (1) AI covers all `ts`, (2) all `ts` are in the area of theory, and (3) the user model indicates that the user has no experience in the field of theory, then AI can be evaluated as high on the scale of being theoretical. In generation, one can use this rule in the other direction: in order to support the conclusion that AI is theoretical, it is sufficient to find three propositions in the knowledge base or user model matching the pattern. When three such propositions can be found, they are added to the pool of propositions that the explanation must convey, as shown in the third chunk of Fig.2-6.

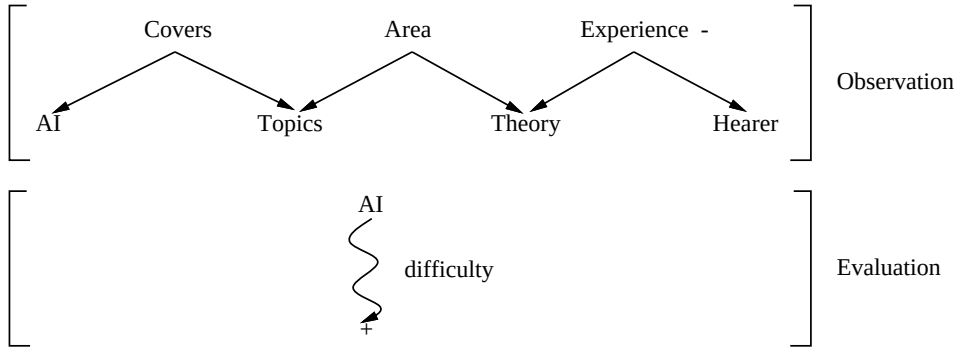


Figure 2-7: An evaluation function

In addition, all propositions are annotated with an argumentative feature indicating that they all serve as arguments for the given conclusion. A proposition extracted from the knowledge base is called an *observation* while I call an annotated proposition a *judgment* (this terminology is derived from [Dispaux 84]). The formal representation of an annotated proposition is shown in Fig.2-8.

There is one topic, logic, in the area of theory which is covered by AI.

```
((cat relation)
 (name topics-of)
 (roles ((class  ((cat class)
                  (name AI)))
         (topics ((cat set)
                  (kind ((cat topic)))
                  (cardinality 1)
                  (intension ((cat relation)
                              (name area-of)
                              (argument {^ roles topic})
                              (roles ((topic ((cat topic)))
                                      (area ((name theory)))))))
                  (extension ((cat list)
                              (elements ~(((name logic)))))))))
 (AO ((cat evaluation)
      (evaluated {roles class})
      (scale ((name theoretical)))
      (orientation +))))
```

Figure 2-8: Functional description produced by an evaluation function

The figure uses the FUF syntax to encode functional descriptions, which are lists of attribute-value pairs. The syntax is fully explained in Chap.3. To understand the figure just note that the `{}` notation indicates pointers to other features within the structure. The figure shows the FUF functional description representation of the following complex knowledge base observation:

```
topics-of(AI, topics1) where
        for all t #r# topics1, area-of(t, theory) and
        topics1 = {logic}
```

Literally, this expression means that there is a set of one topic, logic, such that all topics in the set are in the area of theory and all topics in the set are covered by AI. The most important part of the figure is that this knowledge base description, of category `relation` is also annotated with a feature called `AO` for argumentative orientation. In this case, the AO specifies that the role `class` of the relation is being evaluated on the scale of being theoretical; that is, the absolute simple evaluation `<theoretical(AI), +>` must be expressed when conveying the content encoded in this description. In other words, the linguistic realization component must present this fact as an argument supporting the stated conclusion.

Thus a fact is never passed ''alone'' to the linguistic generation component: it is always annotated with an AO feature, and this annotated structure is called a judgment. Evaluation functions are the tools that map from objective observations to judgments that are context-dependent, gradual and goal-oriented.

I have identified the following eight scales in the ADVISOR II domain:[10]

- Goodness
- Interest
- Importance
- Level
- Difficulty
- Workload
- Programming
- Mathematical

The 15 evaluation functions implemented in the current ADVISOR II prototype, provide a semantics for these eight scales by defining which objective facts, contextual factors and argumentative intentions can activate an evaluation on each scale.

Practically, evaluation functions also serve as the single interface between the knowledge base and the generation system. They therefore implement the translation between the different formalisms used in each system (CLASSIC and FUF-based functional descriptions).

In summary, evaluation functions are important in bridging the gap between argumentative intent and propositional content. The use of evaluation functions also distinguishes this method from previous generation systems in an important aspect: the content derived from the knowledge-base is annotated with argumentative orientation features that indicate its function in the interaction. These annotations are used when phrasing the explanation to select words and constructions that reflect this argumentative function, and allow the lexical chooser to make decisions that would not be possible without this extra information.

2.4.3. Knowledge Sources and their Acquisition

Evaluation functions require the specification of two knowledge sources: a knowledge base and a set of scales on which objects of the knowledge base can be evaluated. I now describe how this knowledge is represented in the system, and how it has been acquired.

Information about the classes and the student is represented in a knowledge base implemented in the CLASSIC formalism [Resnick et al. 90; Brachman et al. 90], a KL-ONE type of language. The knowledge base contains a collection of objects of different types: class, course, student, teacher, topic and assignment are the most important in our domain. Each object is identified by a set of attributes. For example, the class object has for attributes `area`, `prerequisites`, `topics-of`, `assignments-of`, `followup-class`, `required`. Each attribute can be seen as a

[10]The way the scales have been identified is discussed below. I want to thanks Chaya Ochs for her assistance in the identification of evaluation functions and in their implementation.

binary relation between an object and a set of other objects. For example, a `class` object is related to a set of `tasks` through the `assignments-of` relation. This knowledge base is described in detail in Chap.7.

The second knowledge source needed is a set of evaluation scales along which classes can be evaluated. Since the use of such scalar information is quite specific to this approach, I now explain how it can be acquired and modeled. As explained above, the semantics of the scales is defined by the evaluation functions which determine under which conditions a scale can be activated, and by the topoi which determine how the scales are related to each other. The current implementation contains 14 primitive topoi linking the eight scales of the domain. More topoi can be produced by applying transformations to the primitive topoi (*e.g.*, `/+, -/` to `/-, +/`) and by linking topoi.

Scales and topoi have been identified by analyzing a corpus of real advising sessions that has been recorded, and by looking for linguistic clues of scalar reasoning. The analysis was focused on uses of scalar adjectives, vague quantifiers such as *many* or *a lot* and connectives such as *but* or *therefore*. In particular, I performed an exhaustive analysis of the 40,000 word corpus for the analysis of adjectives. In this corpus, I identified approximately 1400 occurrences of 240 distinct adjectives. I focused on all occurrences of adjectives modifying a course, in both predicative and attributive positions and found 150 such occurrences, of 26 distinct adjectives. The next step was to cluster these adjectives in semantic classes, each class providing a first approximation of scale. The results are shown in Fig.2-9.

Figure 2-9 highlights an important property of the domain: a lot of what is said about a class in real advising sessions is scalar evaluations. It also shows that an easy to apply, quite reliable procedure can be designed to identify salient scales in a specific domain. Focusing on adjectives to identify the semantic scales underlying the domain is motivated partly by the technique of the *semantic differential* of Osgood (cf. [Osgood, Suci and Tannenbaum 57, p.20] for a discussion of the use of scalar adjectives for ‘‘meaning measurement’’, cf. also Sect.2.3.2 for a discussion of Osgood’s theory).

Note that a similar procedure of exhaustive corpus analysis and clustering was performed on nouns and verbs in order to identify the classes and relations needed in the knowledge representation of the domain.

Semantic class	Adjective	Occurrences
Difficulty [24]	advanced	1
	basic	1
	challenging	1
	difficult	4
	easy	5
	hard	11
	high-level	1
Domain [8]	mathematical	2
	programming	4
	theory	1
	computing	1

Semantic class	Adjective	Occurrences
Importance [24]	important	10
	needed	1
	recommended	5
	required	5
	suggested	1
	useful	1
	valuable	1
Evaluative [10]	interesting	4
	perfect	1
	good	5
Misc [3]	traditional	1
	new	1
	interdisciplinar	1

Figure 2-9: Adjectives modifying courses in corpus

2.5. The Role of Topoi in Content Organization

Once the content of the answer has been determined and a set of semantic propositions with their argumentative annotations has been produced, the generator must organize and structure the answer. This is the task of the content organization module. I first explain in this section which issues are involved in content organization, by identifying a set of dimensions along which a paragraph structure can vary. I then briefly survey existing work in content organization and finally describe how topoi can serve as the basis of a content organization procedure operating on argumentatively annotated semantic propositions.

2.5.1. The Task of Content Organization

The input to the content organization module is a pool of propositions, annotated with argumentative orientation features, as shown in Fig.2-6. The output is a network where each node is a proposition and each link a rhetorical relation. Figure 2-10 illustrates different answers generated from the same pool of propositions (shown in Fig.2-6), but using different rhetorical structures.

AI covers NLP, a topic that interests you.
SO it should be interesting.
I would recommend it.

AI should interest you,
BECAUSE it offers lots of programming assignments.

AI can be difficult,
BECAUSE it requires a lot of work
AND it can be pretty mathematical,
BUT it is a very interesting course,
BECAUSE it covers many exciting topics,
AND it includes a good number of programming hws.

AI covers nlp, a topic that interests you,
AND it has plenty of programming assignments.
SO, it is a very interesting course,
EVEN THOUGH it can be rather difficult,
BECAUSE you have to write some papers,
AND it covers many theoretical topics.

Figure 2-10: Different rhetorical structures

Figure 2-6 shows that four evaluation functions matched the query. For each match, the evaluation function identifies a set of knowledge-base observations, a judgment (*i.e.*, an evaluation supported by the observations) and a chain of argumentative inferences linking from the judgment to the eventual conclusion. The topoi used in this chain are assumed to be shared by the speaker and the hearer.

Figure 2-10 shows possible rhetorical structures corresponding to this input, assuming that the argumentative intent of the answer is to recommend taking the AI course. In this figure, indentation is used to indicate the subordination relations between main points and elaborations, and connectives expressing the articulation of the answer are printed in all caps. The list shown only hints at the wide variety of options available to the content organization module. To better define the task of content organization, I have identified dimensions along which the structure can vary. Two main dimensions have been characterized: mode of selection of propositions and mode of aggregation. The different options which determine how to map from the content shown in Fig.2-6 to one of the structures shown in Fig.2-10 are summarized in Fig.2-11.

The first dimension concerns selection of content out of the input pool. There are several options: one might always convey all the propositions of the input in the answer, or select only a subset of those. When selecting a subset, different criteria can be used: select a subset of a fixed size (*e.g.*, only one proposition, probably the ''best'' according to some metric), select only the arguments that support the argumentative intent of the answer (as in the first two paragraphs in Fig.2-10) or include also contradictory arguments (as in the last two paragraphs).

Dimensions of variability in content organization

1. Selection:	**2. Aggregation:**
1.1 all	2.1 factoring (factor out redundant info)
1.2 n only	2.2 filtering (redundant and recoverable info)
1.3 + only	2.3 connective
1.4 + and -	2.4 embedding

Figure 2-11: Issues in content organization

The second dimension concerns aggregation and structure. The input in Fig.2-6 is made up of chains of propositions, connected by argumentative relations. The content organization module can factor out common parts of several chains. For example, chains 1 and 3 both end with the same links `/+difficult,-take/`. It is therefore possible to build a tree-like structure, as shown in Fig.2-12. In this tree, the leaves correspond to the observations found by the evaluation functions; that is, to propositions extracted from the knowledge base. Each link corresponds to a topos relation between two evaluations. Given the tree structure, one can avoid repeating redundant information and use conjunction instead to group arguments supporting the same conclusion (as in the last two paragraphs of Fig.2-10). Another way to deal with redundant or recoverable information is to just filter it out and leave it implicit. For example, in the second paragraph, the conclusion that the advisor recommends taking AI is not stated explicitly.

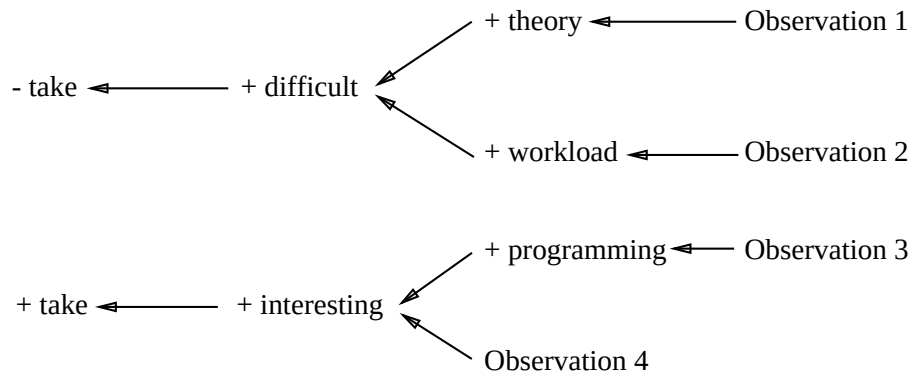


Figure 2-12: Factoring in content organization

When several arcs reach the same node in the tree, as shown in Fig.2-12, the linguistic realizations corresponding to the several conceptual predicates can be combined into a single linguistic constituent. For example, in the top part of the figure, two arcs reach the node `+ difficult` and one arc leaves it. Two methods of combination can be considered: by using connectives, several propositions can be combined (either in a conjunction as in *AI is difficult: it is quite theoretical and requires a lot of work* - or in a subordination as in *AI is difficult because it requires a lot of work*). Often, the connective can be left implicit and the clauses realizing the propositions are just put in sequence. Another method for combining propositions is to embed one proposition as a modifier of one of the noun phrases of a clause. Embedding can be used when two propositions share a conceptual entity that is realized by a noun phrase as in *AI which covers some theoretical topics is quite difficult.*

The task of the content organization module can, therefore, be described as follows: traverse the tree shown in

Fig.2-12 and at each node choose a selection strategy (include the arguments or not) and an aggregation strategy (how to add new propositions to the structure being built). In this work, I have not investigated how this choice is made and what factors influence it.[11] I have instead studied the range of decisions that must be made to structure a paragraph, and most importantly, how these decisions affect the linguistic realization of each clause within the paragraph.

2.5.2. Previous Work on Content Organization

In previous work in text generation, two techniques have emerged to handle the task of content organization: rhetorical schemas and RST (for Rhetorical Structure Theory).

Schemas, introduced in [McKeown 85], encode conventional patterns of text structure. A schema is associated with a communicative goal and describes how this goal is conventionally satisfied. For example, the constituency schema is used to describe the parts of an object, and the process schema [Paris 87] is used to describe a complex process. Such schemas are devised by empirical observation. A schema describes a sequence of *rhetorical predicates* where each predicate is either a primitive communicative function, which can be fulfilled by a single proposition, or recursively another schema. Examples of primitive predicates include *attributive* to attribute a property to an object and *illustration* to elaborate a claim by an example.

When activated, a schema is applied to an input concept from an underlying knowledge-base. Each predicate in the schema is mapped to a query in the knowledge base which is evaluated on the input concept, thus determining the content of each proposition in the paragraph (for example, [McKeown 82, pp54-61] lists the semantics of each predicate used in the TEXT system). The output of a schema application is, therefore, a sequence of propositions labeled by the name of the rhetorical predicates they instantiate. In most implementations, transition words and connectives are also added by the schema traverser and selected before the generation of each clause starts. Schemas, therefore, address at the same time the tasks of content determination and content organization.

When the schema leaves a choice under-determined, other pragmatic factors are taken into account to commit to a choice: for example, in Mckeown's work, focus transition rules derived from Sidner's work constrain the order in which concepts can be referred to [Sidner 79]; in Paris's TAILOR system a user-model is consulted [Paris 87].

While schemas label each proposition as the instantiation of a predicate, RST attempts to label the *relation* between propositions. RST, as introduced in [Mann & Thompson 87], was first a descriptive theory aiming at enumerating possible rhetorical relations between discourse segments. RST relations include *elaboration*, *anti-thesis* and *solutionhood*. A relation connects two text spans, which can be either single propositions or recursively embedded rhetorical relations. One argument of the relation is marked as its ''nucleus'' - and conveys the main point of the text span - while the others are the ''satellites'' (subordinate).

RST was made operational as a technique for planning the structure of paragraphs in [Hovy 88b] and [Moore & Paris 89], elaborating an idea first introduced in [Mann & Thompson 87]. The idea is to attach a communicative intent with each RST relation and to view the combining of relations into paragraphs as a planning process, decomposing a high-level intention into lower-level goals that eventually can be mapped to the utterance of single propositions. The communicative goals associated with the leaves of the structure are then used to determine the content of each proposition. By making the intentional structure of the paragraph explicit, this work combines easily with the planning-based content determination techniques. Note also that, since in RST with planning, the structure of paragraphs is dynamically derived, it is possible to view schemas as the compilation of RST configurations with some information abstracted out, as pointed out in [Moore & Paris 91] or [Hovy 90, p.31]. In addition, [Hovy 91] introduces the notion of *growth point* in RST relations: a growth point is an opportunity for the text planner to attach more information to one of the nodes of the structure being constructed. An example of an annotated RST relation is shown in Fig.2-13. This example shows how the RST relation of sequence is formalized as a plan which can be applied to communicate the information that a `sequence-of` relation holds between two input propositions

[11] The factors determining the choice of a content organization strategy are mainly conceptual: what argument will have the most effect on the hearer, do I need to counter some preconceptions of the hearer, which argument should be given more salience in the structure. I focus in this work on the more linguistic decisions.

(the variables ?PART and ?NEXT). When the relation is selected, a choice of connectives is also determined (no connective, *then* or *next*). Note that this definition crucially depends on the domain-specific relation called NEXT-ACTION which determines when it is possible to view two events as members of the same sequence.[12].

Name: SEQUENCE
Results: ((BMB SPEAKER HEARER (SEQUENCE-OF ?PART ?NEXT)))
Nucleus requirements/subgoals: ((BMB SPEAKER HEARER (TOPIC ?PART)))
Satellite requirements/subgoals: ((BMB SPEAKER HEARER (TOPIC ?NEXT)))
Nucleus+Satellite requirements/subgoals: ((NEXT-ACTION ?PART ?NEXT))
Nucleus growth points:
((BMB SPEAKER HEARER (CIRCUMSTANCE-OF ?PART ?CIR))
(BMB SPEAKER HEARER (ATTRIBUTE-OF ?PART ?VAL))
(BMB SPEAKER HEARER (PURPOSE-OF ?PART ?PURP)))
Satellite growth points:
((BMB SPEAKER HEARER (ATTRIBUTE-OF ?NEXT ?VAL))
(BMB SPEAKER HEARER (DETAILS-OF ?NEXT ?DETS))
(BMB SPEAKER HEARER (SEQUENCE-OF ?NEXT ?FOLL)))
Order: (NUCLEUS SATELLITE)
Relation-phrases: ("" "then" "next")
Activation-question:
"Could ~A be presented as start-point, mid-point, or end-point of some succession of items along some dimension? -- that is, should the hearer know that ~A is part of a sequence?"

Figure 2-13: The RST relation/plan SEQUENCE (from [Hovy 90, p.22])

2.5.3. Argumentation and Content Organization

Topoi can be used for the task of content organization when one notices that topoi can be directly interpreted as rhetorical relations. While topoi are different from RST-like relations, they account for many of the phenomena for which RST relations were invented: they explain the coherence between two text spans in terms of a conceptual relation; they can be related to communicative goals such as argumentative intent and a form of planning can compose several topoi into extended chains. For example, in order to answer whether the student should take AI, ADVISOR II uses backward chaining through the topoi base to link a possible conclusion like +take(ai) to an evaluation function, and eventually to some observations in the knowledge base, to produce the chains shown in Fig.2-6. This backward chaining is similar to the process of top-down planning discussed in [Hovy 91]. In addition, when merging several topoi chains together, to build a tree as shown in Fig.2-12, each merging point plays a role similar to the growth points of [Hovy 91]. So, in the argumentative domain, topoi account for most of the phenomena that RST relations are supposed to explain.

But in addition, topoi also provide further benefits to the problem of content organization, that derive from the two key aspects in which they differ from RST-like relations: topoi are both domain and situation specific and topoi are linked to the lexicon.

Because topoi are domain dependent and user dependent, it is possible to build an operational system that determines whether a relation is applicable in a certain situation of enunciation. In fact, when using RST or a related theory, system implementors must also attach domain-dependent predicates to each generic relation to determine when it can be used (for example, the NEXT-ACTION relation in Fig.2-13). So while the theory of RST claims to have uncovered generic rhetorical relations, each implementation must redefine what it means to be an example, what it means to be a solution etc. in domain specific terms. In [Mann & Thompson 87], this task is described in

[12]The notation BMB is the ''mutual belief'' operator: (BMB S H P) means that P is mutually believed by S and H

terms of ‘‘oracles’’ - black boxes capable of answering questions like *if I say X, will the reader believe it?*. But RST provides little guidelines to build such oracles in practical systems. And indeed, implementors are often left struggling, trying to find operational nuances between RST relations like *cause*, *reason*, *justification* and *evidence* (all four relations are used, for example, in [Horacek 92]). In contrast, when using topoi as a primary form of rhetorical relation, one does not need to distinguish between the generic level and the domain dependent. There is no need to develop a vocabulary like *evidence*, *justification* etc. One can implement a text planner where the domain relations themselves can be used to account for the rhetorical structure.

The second aspect in which topoi differ from RST-like relations is that topoi are related to the lexicon and to the rules of syntax, as is detailed in Sect.2.6 (cf. also [Elhadad 92] for a development of this argument). Therefore, when selecting a topos, the system not only creates a complex text span (a text structuring operation), but also constrains the surface realization of this text span. Because topoi annotations are kept with each proposition sent to the lexical choice module, the rhetorical function of a text segment can have an effect on all aspects of its realization. In contrast, in an RST-based text structurer, the position of a text span within the rhetorical tree only affects the choice of connectives, as indicated by the `relation-phrases` feature of Fig.2-13 (this limitation is partially acknowledged in [Hovy 91, p.96]). But the rhetorical function of a proposition affects its surface realization at many levels. Consider the following utterances, extracted from a corpus of real advising sessions:

It requires quite a lot of programming
It does involve some programming, but nothing outrageous.

Both sentences can be generated from the *same* observations in the knowledge base, but the difference between the two forms is determined by the argumentative function of the clause. This argumentative difference explains the selection of *a lot* vs *some*, *require* vs. *involve*, of the concessive construction *does include ... but ...*. In that sense again, topoi can serve as a bridge between early decisions in the generation process (content organization) and later decisions (lexical choice).

2.6. The Role of Topoi in Lexical Choice

The task of the lexical choice module is to select which words will eventually appear in the answer. First, a brief overview of the issues involved in lexical choice is presented. I then focus on how argumentation can be used to help perform lexical choice, and show how a verb conveying an appropriate connotation is represented in the lexicon and how it gets selected by the generator.

2.6.1. The Task of Lexical Choice

Input to the lexical choice module is the rhetorical structure built by the content planner; that is, a network of conceptual propositions that are annotated by their argumentative function. A comprehensive survey of research in lexical choice is provided in [Robin 90]. I enumerate here some constraints that have been identified on lexical choice in previous work.

There are two aspects to the lexical choice task:

- The lexical chooser selects lexical heads out of the conceptual network, and for each head, determines its syntactic category and which syntactic constituents depend on the head.
- The lexical chooser selects an entry from the lexicon to map each head to an actual word.

The first aspect interacts closely with the syntactic decisions to be made when generating a sentence. In previous work, this was done by associating predicate-argument relations of the conceptual network with phrasal templates. For example, [Danlos 87a] relied on a lexicon-grammar and [Jacobs 87] on pattern-concept pairs to perform this task.

The second task of a lexical chooser is, given a conceptual head and its dependents, to choose an appropriate lexicalization. There is indeed a many-to-many mapping between conceptual entities and relations and the lexical entries. There are several ways to choose between different lexicalizations. In certain cases, the choice has no

impact on the appropriateness of the generated text. For example, the node `AI` can be realized by *AI*, *the AI course*, *Artificial Intelligence*. In most situations, the choice between these alternatives has little impact. In other cases, the difference between alternative lexicalizations can be related to stylistic factors: length of the expression, formality etc. Such factors have been used in [Hovy 88a] for example.

In yet other cases, alternative lexicalizations can be determined by the semantics of the arguments of the lexical head. For example, if the conceptual entity is `produce-noise`, then depending on the semantic type of the agent, the words *bark*, *resound*, *meow* etc... would be selected. Such selections have been handled in previous work by building discrimination trees, which describe how to progressively refine a generic concept into a specific lexicalization (cf. [Goldman 75] for the original implementation of this idea and [Danlos 87a; Buchberger & Horacek 88] for recent uses of the technique).

Finally, in many cases, I have also found that one can choose between alternative lexicalizations by looking at the argumentative intent of the utterance. I have presented some of the issues involved for connectives in [McKeown & Elhadad 91; Elhadad & McKeown 90], for adjectives in [Elhadad 91b], and for the selection of verbs and adverbials in [Elhadad & Robin 92]. Details on these lexical classes and similar principles for the selection of determiners are presented in Chap.6.

2.6.2. Using Argumentation for Lexical Choice

There are two demands a generator must meet when mapping a conceptual predicate-argument structure to a linguistic structure: completeness - each conceptual predicate-argument must be ‘‘covered’’ by a linguistic predicative relation - and coherence - no unwanted semantic information should be conveyed. For example, the relation `assignments-of` between the entity `AI` and the set `HWS` in Fig.2-5 is ‘‘covered’’ in the sentence *AI has many assignments*, but not in the sentence *John takes AI and does the assignments* (it is therefore not a complete rendering of the input) and the proposition `DO(John, Hws)` is expressed by the second sentence while it is not part of the input (it is therefore not a coherent rendering of the input).

Similarly, argumentative evaluations present in the input must be ‘‘covered’’ by some linguistic marker to be satisfied. In ADVISOR II, the use of topoi in content determination and organization has produced argumentative information and attached it to the input to the lexical chooser. The task I now consider is how to satisfy these new demands on the lexicalization module: how to choose words that both convey a given semantic information and express an argumentative intent.

In the work on lexicalization and surface generation presented in this thesis, I have looked at how the following decisions can be made non-arbitrarily by the generator, and motivated by argumentative factors:

- Choice of scalar adjectives in the domain (words like *interesting* or *difficult*).
- Choice of vague quantifiers (words like *many* or *few*).
- Choice of argumentative connectives (words like *but* or *so*) between clauses.
- Selection between quasi-synonyms that differ only by their connotations

I have developed and implemented ADVISOR II, a complete generation system that produces sentences all the way from a formal query and that includes a comprehensive grammar of English. A complete account of the method proposed for lexicalization is given in Chap.6. To illustrate how argumentation influences lexical choice, the focus in this chapter is on the last one of these choices: the selection of a lexical entry differing from other close synonyms by its connotations.

As an example, consider the contrast between the following sentences:

AI requires/necessitates/demands many assignments.
AI has/includes/involves many assignments.

All six verbs shown express (more or less successfully) the conceptual relation between a class and its assignments, roughly translated in language as a possessive relation. As discussed in Sect.2.3.2, there is, however, a difference

between the first group and the second: *require*, *necessitate* and *demand* all carry an argumentative connotation. In addition to conveying the conceptual relation of `assignments-of` between `AI` and a set of assignments, these verbs also express an evaluation of `AI` as a difficult class. Thus, choosing one of these verbs is sufficient to realize this argumentative evaluation when it is present in the input to the lexical chooser.

Similarly, one can contrast between the following sentences:

You enjoyed AI.
You took AI.
You struggled with AI.

The verbs *enjoy*, *take* and *struggle* all cover the relation that the student has taken the AI class. But *enjoy* and *struggle* also express an argumentative evaluation of the class (as an interesting or difficult class).

Information on such argumentative connotations is stored in the lexicon so that the lexical chooser can consider these connotations as an additional resource to cover argumentative evaluations in the input. I now look at how this information is expressed and used by the lexical chooser.

I use a lexical formalism inspired by HPSG [Pollard & Sag 87] in the implementation, using FUF, the version of functional unification formalism developed for this research (cf. Chap.3). The information in this formalism is uniformly represented by sets of attribute-value pairs called *functional descriptions* or *FD*s. In the following figures, curly braces indicate a disjunctive value. For example, in Fig.2-14, the value of the `lex` feature of the top node can be either *have*, *include* or *involve*.

Figure 2-14 shows how in the lexicon the semantic relation `assignments-of` is mapped to the choice of a verb, when argumentation plays no role. Each lexical head in the formalism has a feature `SemR` (for semantic representation) which contains a partial description of the semantic content covered by the word. For generation, lexical entries are indexed by conceptual objects - classes or relations. Thus, in Fig.2-14, the entry is indexed by the `[Relation = assignments-of]` feature. This relation is mapped to a lexical head of syntactic category `verb` and of type `possessive`. In addition, the entry specifies how the two entities of the semantic representation are mapped to the lexical arguments of the verb, the constituents labeled `owner` and `owned`. This mapping is indicated by the `SemR` connections between the linguistic subconstituents and the semantic arguments.

The mapping shown in Fig.2-14, therefore, expresses a phrasal pattern of the form:

```
Concept:                  assignments-of(Class, Assignments)
Linguistic Structure:     possessive(Owner, Owned)
Phrase:                   Owner {have, include, involve} Owned
Mapping:                  Class->Owner, Assignments->Owned
```

Figure 2-15 indicates how argumentation is represented in the lexicon. It shows the more complex example of the lexicon entry of the verb *to require* with its argumentative connotation. The figure actually shows how the lexical entry is instantiated when applied to the semantic input corresponding to the composite evaluation in the right part of Fig.2-5. The top part of the figure is very similar to Fig.2-14. It describes how the verb *require* covers the semantic relation of `assignments-of`. The only difference is that the verb *require* is not a member of the class of *possessive* verbs, which share many properties and are known to the grammar. Instead, its syntactic properties are all represented in the lexicon (but not shown in the figure). This is indicated by the feature `[type lexical]`.

The other addition is that both of the semantic arguments of the relation `assignments-of` are now the object of a composite evaluation, of the form */the more assignments, the more difficult/*. This evaluation is a composite evaluation, which I write down as:

```
<cardinal(assignments), + <difficult(AI), +>>
```

The `argumentation` feature of the `assignments` argument specifies that it is evaluated on the cardinality scale with orientation +. In addition, the feature also points to the RHS (right-hand side) of the composite evaluation, the embedded evaluation of AI which determines the value of this composite evaluation. The `(CorrelationR +)` feature indicates that the composite evaluation is of the form `/+,+/` or `/-,-/`.

The lexical entry for *require* indicates that the choice of the verb covers the argumentative evaluation of the semR of the complement 1. This relation is indicated by the dashed line labeled `covers` in the figure. Because this

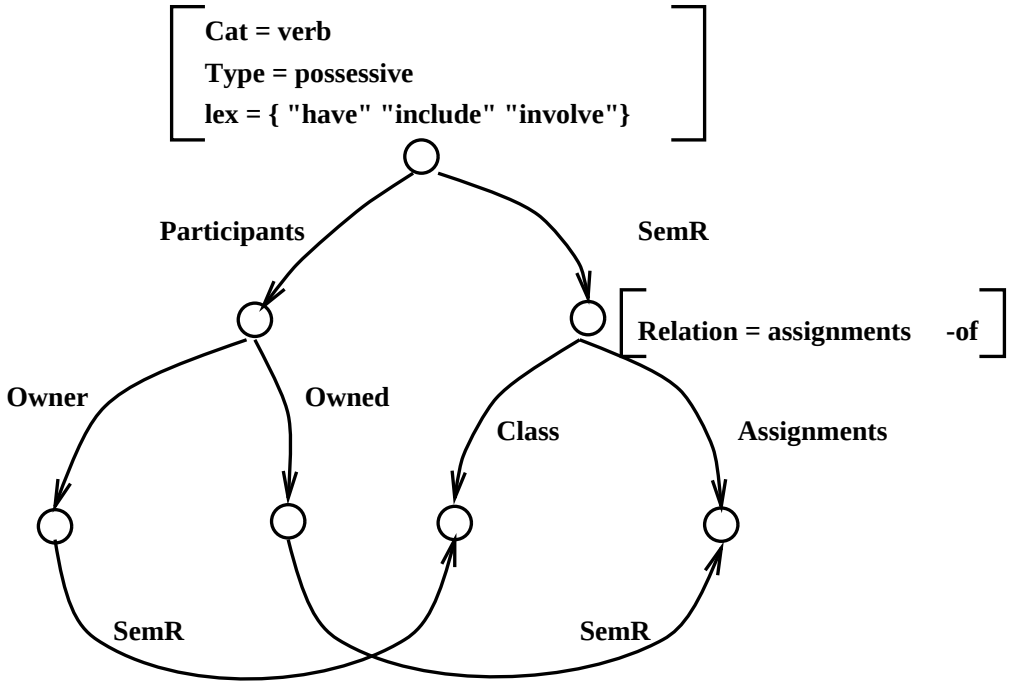


Figure 2-14: A simple lexicon entry for the `assignments-of` relation

evaluation is covered by the verb, no other linguistic device is needed to convey it. In contrast, the evaluation of the `assignments` argument still needs to be covered after the verb has been selected. This can happen when the complement 2 is lexicalized, by choosing, for example, a determiner like in *many assignments*. The `covers` links constitute the principal technique by which linguistic decisions are related to semantic constraints. The same links are used throughout the grammar and lexicon and account for many decisions made by the generator.

In summary, this section has shown how the use of argumentative features can help in the task of lexical choice by accounting for the difference between lexical entries that differ by argumentative connotation while covering the same semantic content (*e.g.*, choice between *include* and *require*).

2.7. Conclusion

This chapter has shown how the theory of argumentation in language helps in the task of text generation. The approach developed in this work relies on a view of language where all decisions, from the high-level choice of speech-acts to the most detailed lexical distinctions can be related to the intentions of the speaker, a view very close to the model advocated in [Fillmore, Kay & O'Connor 88, p.534] and underlying the theory of argumentation in language. I have proposed an architecture where the same abstract formalism can be used throughout the generation process. As a consequence, the same intention can be satisfied by devices at different levels. For example, to convince the hearer that AI is difficult, all the following decisions can be made:

- Include the information that AI has many assignments. *(a content determination decision).*
- Express the relation between number of assignments and difficulty through a rhetorical relation such as ''AI has many assignments, *so* it is quite difficult''. *(a content organization decision)*
- Choose a verb conveying an evaluation of the AI class like *require* in *AI requires many assignments*. *(a lexicalization decisions)*

All these decisions are motivated by the same original speaker intention and can interact with one another.

I have shown how the theory of argumentation in language and the descriptive tool of topoi can provide the uniform

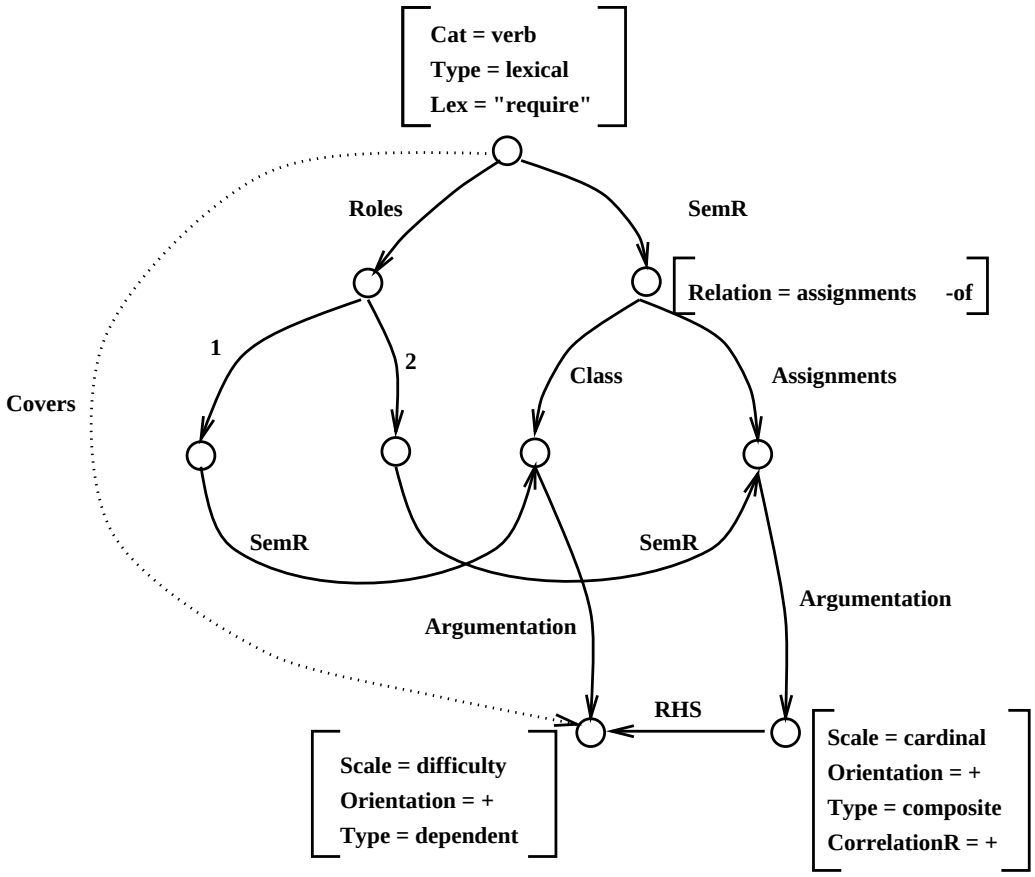


Figure 2-15: A lexicon entry for the verb 'to require'

representation device needed to implement such an architecture. Topoi effectively serve as a bridge between the conceptual decisions and the linguistic decisions made by the generator. Topoi have been found to be abstract and expressive enough to be used for content determination, content organization and lexicalization.

Chapter 3
Functional Unification and Surface Generation

The task of generating language differs significantly from that of interpreting language. In generation, the main issue is to make choices between alternative ways of expressing a semantic content or satisfying a communicative goal. In interpretation, the problem is to recognize a structure and infer from a recognized pattern the intent of the speaker. The difference is thus between controlling choice, on one hand, and recognition and inference on the other hand. Formalisms for generation and interpretation have therefore been developed with different perspectives and problems in mind. The main issue for a generation formalism is representing choice points and controlling multiple interacting decisions. Besides this difference, one important commonality between interpretation and generation is the *compositional* aspect of the task. In generation, communicative goals are fulfilled by combining together a set of linguistic devices according to grammatical rules. Similarly, in interpretation, grammatical rules specify how the observed linguistic elements combine into a coherent structure, from which the interpretation is inferred. Knowledge of such composition rules (the grammar of a language) is required for both generation and interpretation.[13]

The previous chapter has outlined an architecture for a generation system where a single source of knowledge - argumentative relations known as topoi - has an influence on all decisions made during the generation process, from content determination to surface realization. I present in this chapter and the following a generation formalism, called FUF, for Functional Unification Formalism, which is an extension of Kay's Functional Unification Grammar (FUG) formalism [Kay 79]. FUF uniformly represents all the knowledge sources controlling the generation process (topoi base, lexicon and syntactic realization grammar). The main orientation of the FUF formalism is to encode the choice points a generator must consider and how the decisions made a different stages of the generation process interact. Because FUF is used uniformly at all stages of the architecture presented in Fig.2-3 (p.20), content determination, lexical choice and syntactic realization, the decisions made at each stage can interact with one another. Thus, for example, the speaker's argumentative intent can be satisfied by selecting a certain piece of information, choosing words with the appropriate connotations to phrase it, and organizing a sentence in such a way as these connotations clearly appear to the hearer.

FUF controls well the interaction between these decisions because it is a unification-based formalism. Indeed, in FUF, all information is uniformly encoded in data structures called *functional descriptions* or FDs. FDs are sets of *features*. Both the input and the knowledge sources, grammar, lexicon and topoi-base are represented as FDs. FDs can be intuitively interpreted as *constraints*. The input FD is a constraint on what the speaker wants to say. The grammar and lexicon encode constraints on how things can be said and combined together into syntactically correct sentences. The only operation defined on FDs is *unification*. Unification, in this context, is best understood as the operation of merging constraints from the input with those of the grammar and/or lexicon. The result of the unification of an input with a grammar is a new FD, which includes both constraints on what to say and on how to say it. This total FD is thus the description of a linguistic object which satisfies the input constraints. This description can be interpreted by a linearizer which translates it into text.

A choice point in the grammar or in the lexicon is represented by a disjunction of FDs. During unification, the generator must pick an FD out of each disjunction so that the total FD remains consistent. When a branch of a disjunction is selected, *i.e.*, when a decision is made, the corresponding FD is merged into the total FD, enriching it with new features. Thus, unification proceeds both as search through the system of choice points encoded in the

[13]It is this aspect only of linguistic knowledge that reversible grammars attempt to capture (cf. for example [Shieber 88] and [Dymetman & Isabelle 88]). It is important to realize, though, that this aspect is only a subset of the total knowledge required to perform generation: reversible grammars do not specify how the choice between alternative decisions in the grammar relate to the speaker's communicative goals.

grammar and lexicon, and as construction of the total FD, merging in new features as decisions are made. In other words, making choices and building a linguistic structure are both handled simultaneously by the unification mechanism.

FUF is derived from Kay's original FUG formalism [Kay 79]. FUG has been identified in previous work as a well-adapted formalism for surface generation [McKeown 85; Appelt 85; Paris 87]. A major contribution of the work presented in this dissertation is the development of the first large scale portable generation component using a FUG-like formalism, the SURGE grammar. The lessons derived from the development of SURGE have led to the definition of desirable extensions to the original FUG formalism and the design of a new extended formalism called FUF. These extensions and their motivation are presented in the next chapter.

This chapter presents the original FUG formalism and its use for surface realization. While it presents little original material, it provides the most detailed presentation of the FUG formalism available to date and background material not available elsewhere. The mechanisms and notations defined in this chapter are used throughout the rest of the dissertation.

In the rest of the chapter, I first elaborate on what is the role of a formalism for text generation and list constraints a generation formalism must fulfill. Three established generation formalisms are then compared along these lines: FUGs, NIGEL and MUMBLE. All three formalisms have only been used previously for syntactic realization. I conclude from the comparison that FUG is the only formalism which can meet the goal of being extended to handle more of the generation tasks in a uniform manner: content determination and lexical choice. The extensions required to allow such new uses lead to the definition of FUF, presented in the next chapter.

I then proceed to a short tutorial/overview of the use of FUGs for generation. A small toy grammar is presented and the major features of the formalism are illustrated on the generation of a complete sentence.

A complete technical description of the FUG formalism follows, with a systematic review of all the constructs of the FUG language. Functional descriptions are first presented and defined precisely. Special *meta-FDs* are then discussed. The way disjunctions encode choice points is then explained. Paths, which introduce structure sharing in FDs, are then presented and a different analysis of FDs in terms of graphs is discussed. The specification of ordering constraints in linguistic objects using the pattern construct is then introduced. The unification procedure is then presented in detail, with a pseudo-code listing of the main unification functions. Finally, the encoding of linguistic structure and the notion of constituent is introduced.

To further characterize the computational expressiveness of FUGs, an example of using FUGs for a general programming task is then presented. I describe how lists can be encoded as FDs and how the append and member operations can be programmed in FUGs. This exercise serves to introduce a comparison between FUGs and PROLOG and to discuss features of FUGs as a general programming language. This discussion permits to better understand why FUGs are appropriate to the generation task. This last section presents a new and original perspective on FUGs as a programming language, and demonstrates the use of FUGs for applications which were previously thought beyond their range of applicability.

In conclusion, I evaluate the strengths and limitations of the original FUG formalism for text generation.

3.1. Background: Formalisms for Text Generation

3.1.1. Criteria on the Selection of a Generation Formalism

The task of generating language requires the manipulation of complex data-structures: representation of linguistic constituents, lexical entries, their semantic denotation, the links between linguistic objects and the semantic constraints they realize. In addition, generation consists in choosing between different linguistic devices to satisfy a communicative goal, and building a linguistic structure combining these various devices into a coherent sentence or paragraph. It is, therefore, important to develop specialized formalisms to manipulate such complex data in practical implementations, to encode the choices that a generator must consider, to control how choices are made and interact

with one other, and to specify how a linguistic structure can be composed from a collection of smaller constituents. The combination of the emphasis on choice and decision making and on the knowledge required to compose linguistic objects is specific to the task of generation.

In particular, language generation includes both a content and a linguistic component, as discussed in the architecture survey in Sect.2.2. One of the motivations of this work on generation formalisms is to design a uniform formalism to handle all generation tasks of this architecture and thus allow a smooth interaction between decisions made at different stages of the generation process. As a consequence, such a general generation formalism must support:

- the manipulation and description of linguistic structures and constraints upon them.
- the manipulation and description of conceptual structures and constraints upon them.
- the description of the mapping between conceptual and linguistic structures and how this mapping can be constrained.

Each of these three constraints imposes different sets of criteria on what an appropriate formalism for text generation should be, both in terms of expressiveness and efficiency. This section presents the demands that a generation formalism must meet.

In the following paragraphs, I list criteria that can be used to determine the appropriateness of a formalism to the task of generation. These criteria are further discussed afterwards:

- *Description of conceptual input:* Input to a surface language generator should be semantic, or pragmatic, in nature. The formalism should therefore allow the specification of semantic and/or pragmatic features.
- *Partial input specification:* some flexibility should be allowed in what must be provided as input. The generation of a sentence involves a potentially huge number of pragmatic and semantic features. Not all pragmatic or semantic features to which the grammar as a whole is sensitive may always be available for each input concept and the generator should be able to function in their absence by providing appropriate defaults.
- *Input expressiveness:* Input should be general enough to be easily interfaced to several underlying systems with different knowledge representation schemes (therefore allowing the development of portable generation systems).
- *Mixed input*: Few linguistic details should be specified in the input as these should be filled in by the generator, which contains the lexical and syntactic knowledge for the system. It should also be possible, however, to include linguistic constraints in the input, for example, to force one word to be used, to constrain one constituent to appear before another one, or to limit the length of the generated text.
- *Functional description of linguistic objects*: It should be possible to express the function of linguistic objects; that is, to specify how linguistic objects relate to conceptual objects and how they are functionally related to other linguistic objects.
- *Structural description of linguistic objects*: It should be possible to express the structure of linguistic objects; that is, to identify constituents and to specify how constituents can be merged together. It should be possible to characterize the strings generated by a grammar in the formalism so that only grammatically acceptable strings can be generated (no over-generation).
- *Support for compositionality:* It should be possible to link the construction of the linguistic structure with the structure of the conceptual input, and to keep track of this mapping.
- *Declarativity*: constraints on how linguistic objects combine and relate to the input should be expressed declaratively.
- *Order of decision making:* The order in which decisions must be made and the interactions between them has an impact on representation of constraints. If decisions must be made in a fixed order, representation of constraints on those decisions may become more complex. The order of processing, bottom-up, top-down, left to right, or any other variation, can significantly influence how constraints interact.

- *Efficiency:* the formalism should be constrained enough to allow for the definition of efficient constraint solving procedures.
- *Reversability:* Ultimately, a natural language system that uses only one grammar both for parsing and generation is desirable, as the grammar serves as the single repository of linguistic knowledge. Most formalisms have usually been developed with one task or the other in mind. Due to the differences in focus between the two tasks, when a formalism is adopted for the other task, the match is often not ideal. For example, when FUG is used for interpretation, an additional, rather complex chart must be supplied [Kay 85]. On the other hand, when grammars originally developed for interpretation are used for generation (ATNs, DCGs), it has been found that many desirable features are missing (cf. [McKeown & Elhadad 91] for a discussion).

3.1.1.1. Input Specification

The first four criteria relate to the specification of the input to the generator. The task of the generator is to map an input specifying pragmatic and semantic input onto a linguistic object. The criterion demanding most explanation is the one requiring the possibility of a mixed input; that is, input specifications must be allowed to contain both pragmatic/semantic features and linguistic constraints. This constraint relates to the following critical observation: generation should not only be viewed as a rewriting process mapping from purely semantic input to linguistic output. Indeed, one main task of a language generator is to make decisions about the syntactic structure and the lexical items to use. But, because the linguistic system has a structure of its own, whenever a single decision is taken by the generator to satisfy a semantic input constraint, the decision in turn creates additional constraints on which other linguistic elements can co-occur with it. For example, if the generator decides to lexicalize an action by using a verb in a declarative clause, then it must also find a subject for the verb (because English syntax requires declarative clauses to have a subject), and in turn, the subject and the verb must agree in number. The constraints to include a subject and to perform the agreement are not motivated by the input; they simply derive from the combination of one isolated decision with the linguistic system of English. Furthermore, certain input constraints may specify which linguistic forms should be used (style imposing constraints on length of expression, on use or non-use of certain words, or even formal constraints like rhyming). A generation formalism should therefore allow the specification of linguistic constraints in the input, mixed with the semantic message to be conveyed, and allow the grammar to add purely linguistic constraints as they arise because of the rules of the linguistic system.

3.1.1.2. Functional vs. Structural Perspectives

The next two criteria relate to the distinction between the *functional* and *structural* perspectives on language. In the structural analysis, linguistic objects are considered as strings of signs, substrings of these strings are recognized as members of a certain class of constituents; different rules specify how the concatenation of segments of different categories can create larger constituents. For example, the concatenation of a string of category *article* with a string of category *noun* could create a string of category *noun-phrase*. Certain formalisms allow the use of other combination techniques to merge subconstituents (*e.g.*, head wrapping in GPSG [Gazdar et al 85; Sells 87, Sect.3.5], adjunction in TAGs [Joshi 87]). The output of a structural analysis is a parse tree. The structural perspective focuses on surface order of subconstituents in a linguistic object and on the formal characterization of the set of strings that are part of the language.

The purpose of a functional analysis is to identify the *function* of each constituent within a larger sequence of words. There is also a structural stage in a functional analysis, whose purpose is to identify the *constituents* of the sentence. A constituent is identified as such only if it plays a certain function with respect to the higher level constituent containing it. For example, in *John gives a book to Mary*, the group *a book* forms a constituent, of category *noun group*, and it plays the role of the *object upon which the action is performed* in the clause. (This role is often called the *affected* role in functional grammars.) The functional perspective focuses on the mapping between linguistic objects and an *informational domain*; that is, a level describing the purpose of language. The functional perspective relies on a structural analysis to relate the structure of the informational domain to the structure of the linguistic objects (a form of compositionality).

The language generation task requires both aspects of knowledge of language - structural and functional - to be represented: a generator must only produce grammatically correct text, and a generator must only produce text that satisfies a communicative intent given in input. An appropriate generation formalism must therefore provide

support for both perspectives. There is, however, a greater emphasis in generation on the functional perspective than in interpretation. The main reason is that in interpretation, things start with a structural analysis: until constituents have been identified, no further interpretation can proceed. In contrast, in generation, things start with a functional analysis: the input to the generator consists of a functional description of semantic information or of a communicative goal. Emphasis in general is therefore put in generation on the functional perspective.

3.1.1.3. Declarative vs. Procedural

The next criterion to elaborate is the preference for *declarative* formalisms over *procedural* ones. A procedural formalism specifies precisely how mappings are computed between conceptual and linguistic structures. In contrast, a declarative formalism only specifies constraints on realizable mappings, and relies on some form of constraint satisfaction mechanism to establish a specific mapping. Because many influences, from various sources, contribute to the generation of even a single clause, procedural formalisms often require too much complexity. For example, lexical choice is constrained by several interacting pragmatic factors, lexical relations between the words of the sentence such as collocations [Smadja 91a], semantics of the domain and syntactic properties of the words. Taking all these constraints into consideration into a single procedure creates complex and inflexible generators. In general, to avoid the complexity, procedural generators decompose the generation process into a sequence of manageable tasks (this is the strategy used in MUMBLE for example, as explained below). But the consequence of this approach is that order of decision making is then fixed and that decisions in each task cannot interact with decisions in the other. In general thus, declarative formalisms allow for more flexible and more easily manageable generators.

3.1.1.4. Uniform vs. Specialized Formalisms

Another issue concerning the choice of a formalism is the choice between a uniform formalism for as many tasks as possible as opposed to a collection of specialized formalisms for each task. In general, a generation formalism is torn between the demands of underlying systems which provide the information to be conveyed in language and the demands of the linguistic system. Some researchers argue that different formalisms should be used for the conceptual and linguistic components for example (cf. [McDonald et al. 87, p.170]). In contrast, I follow a model of linguistic theories advocating a functional perspective where conceptual objects can be described by sets of associations between features and values, known as *feature structures* (cf. for example [Shieber 86], [Johnson 88] and for specific linguistic theories adopting this perspective [Pollard & Sag 87] on HPSG, [Kaplan & Bresnan 82] on LFG). Kay has been among the first to propose using the same formalism to encode both syntactic and semantic information (cf. in particular [Kay 84]), and this suggestion has had important ramifications in the development of HPSG for example. In this work, the main motivation for using a uniform formalism is that I have found that the same data structures (*e.g.*, the topoi discussed in Chap.2) can play a role at both the conceptual and linguistic levels. I have therefore focused on developing a single formalism that can be used throughout the generator, to allow for an easy interaction between the decisions made at different stages of the generation process.

3.1.1.5. Complexity, Performance and Expressiveness

The last dimension sharply dividing existing formalisms concerns efficiency. The choice is between expressive and potentially inefficient formalisms and less-expressive formalisms with guaranteed worst-case behavior. FUGs provide an extremely expressive formalism for generation. It is however certainly too expressive and that comes with a cost: the FUG formalism has been proven to be capable of describing arbitrary constituent structures, and the problem of determining whether an arbitrary FUG generates an arbitrary string (*i.e.*, parsing a string with a FUG) is undecidable (cf. [Johnson 88, pp.87-95]). This theoretical result has of course consequences. Many researchers are actively pursuing work trying to define a formalism that is less expressive and more constrained than FUGs. Most notable is the search for so called ‘‘mildly context sensitive’’ formalisms illustrated by TAGs (tree adjoining grammars) [Joshi 87]. Other researchers have advocated the use of deterministic models that make indelible decisions as a psychologically more valid model of language production [Meteer et al. 87; De Smedt 90]. But there are reasons why using a formalism more expressive than necessary is practically productive:

- The class of grammars written in FUG can be restricted to a computationally decidable subset (cf. [Johnson 88, pp.95-102] and the ‘‘off-line parsability criterion’’).
- As long as a practical implementation is fast enough to experiment on a certain class of grammars, the worst case results of a theoretical analysis are not of concern.

- I am not interested in modeling speech production but written text production, which is characterized by a number of revisions, and for which no psychological model is readily available.
- Superior expressiveness is important when experimenting with a grammar. Only when a specific grammar turns out to be necessarily too complex computationally, does it need to be revised. Converting a grammar into a formalism with guaranteed acceptable worst-time behavior can be seen as a sort of optimization applied to a specific grammar.

There is in general a trade-off between expressiveness and efficiency. I have opted for the expressiveness side in this work, and have found that practical engineering can yield systems that produce sentences sufficiently fast for interactive use. I have developed a practical unification system and used it as the basis of this research. The system is called FUF and is implemented in Common Lisp. Currently, it takes less than two seconds for FUF to generate a sentence on a Sun SparcII workstation (1991 class workstation, 28 MIPS). Chapter 4 presents a set of extensions to the FUG formalism that have allowed FUF to reach this level of performance on a complex grammar of English, and this experience with practical grammar writing justifies the approach adopted in this work.

3.1.2. Existing Generation Formalisms

Different choices can be made to satisfy the different demands on a formalism listed above, and several generation formalisms coexist. I use in this work FUF, a unification-based formalism derived from *functional unification grammars* (FUGs, [Kay 79]). I have developed using FUF, a large portable surface generation component called SURGE which has been used as the generation module of several systems (ADVISOR II described in this dissertation, COMET in [McKeown et al 90; Elhadad et al. 91], COOK in [Smadja 91a], STREAK in [Robin 92a]). FUF and SURGE are also being used at more than 20 research sites outside of Columbia. Two other portable systems have been developed prior to FUF: NIGEL [Matthiessen 85; Mann & Matthiessen 83b] and MUMBLE [McDonald 80; Meteer et al. 87]. These three systems represent three different perspectives on what facilities a generation formalism should provide. I now briefly review each of these three formalisms.

I only focus in this section on implemented formalisms that have been designed specifically for surface generation and ported to several applications. It is important to distinguish between computational formalism and grammatical theory, because the two notions often get blurred. The best example of this blur is offered by the case of systemics. Systemics is a linguistic theory, that determines what is the object of linguistic analysis, what are observables and what is to be explained by linguistics. One of the internal assumptions of the theory, however, is that a certain computational formalism provides a particularly perspicuous representation of linguistic knowledge. This formalism is centered around the notion of *system* [Halliday 76a]. Although this formalism was not conceived as a computational formalism, it has all the attributes of a computational formalism (cf. [Patten 86] and [Brew 91] for such an interpretation). Similarly, recent linguistic theories use unification and feature structures as a formalism to encode linguistic knowledge (cf. for example HPSG [Pollard & Sag 87, Chapters 2 and 8]). But unification-based linguistic theories are foremost linguistic theories, that offer a substantial message about language. Thus, HPSG and LFG are distinct linguistic theories, even though both are unification-based.

Even though linguistic theories often seem committed to a particular formalism, it is important to abstract the substantial linguistic theory from the formalism. Thus, it is possible (and I believe beneficial) to encode systemic grammars in a unification-based formalism. Similarly, different unification-based linguistic theories are committed to different flavors of unification formalisms, but most can be encoded in a generic unification formalism such as FUGs.[14] In this section, I compare computational formalisms, not linguistic theories.

3.1.2.1. FUG

Functional Unification Grammars (FUGs) were introduced by Martin Kay in 1979 [Kay 79]. FUG Is an instance of the unification-based formalisms becoming prevalent in linguistics (cf. [Shieber 86] for a survey of the field). It has had particular success in the generation field particularly with its adoption in work by McKeown [McKeown 85] and

[14]Actually, it appears that most recent unification-based linguistic theories have been heavily influenced by Kay's FUG formalism and the idea of representing syntax and semantics uniformly in feature structures he presented in [Kay 84] (cf. [Pollard & Sag 87, p.49-50] for a discussion).

Appelt [Appelt 85]. FUG has also been proposed for parsing [Kay 85], but has never been used extensively for that purpose. The explanation for this mostly generation orientation is most likely that FUG puts more emphasis on the functional than on the structural perspective on language. Since generation starts with functional considerations (how to map meaning to form) and ends with structure (how to organize form), the mostly functional perspective of FUG has provided a good framework for generation. In contrast, parsing with FUG requires the extraction of complex charts out of the grammar, which makes the process less transparent and easy to use.

Note that I present here the original FUG formalism. FUF is both an implementation and an extension of FUGs. The implementation aspect (which includes making certain aspects of the abstract formalism more precise) is presented in the rest of this chapter. The extensions are presented in the following chapter.

Of the three formalisms presented in this section, FUG is the one that is least bound to a particular grammar system. In fact, FUG can be used to express grammars of very different linguistic schools: I have implemented in FUF grammars based on phrase structure rules, systemic grammars and HPSG-like grammars. Kasper reports on experiments in converting the NIGEL grammar into the FUG formalism to construct a parser with equivalent coverage [Kasper 88]. So FUG is more a computational formalism than a grammar system. It does not impose a particular substantive perspective on linguistics.

Since the rest of this chapter presents FUGs in detail (through the FUF implementation), only the distinctive characteristics of FUGs are listed here:

- *Uniform representation*: FUG relies on a single data-structure called the *functional description* (FD) to encode uniformly syntactic, semantic and pragmatic factors. FDs capture both functional and structural aspects of a linguistic description. They are also expressive enough to serve as general purpose programming constructs[15] and therefore can be interfaced with a variety of knowledge representation systems. Figure 3-1 shows an example FD encoding the input for the sentence *John gives a blue book to Mary* to the grammar SURGE.[16]
- *Unification*: FUG relies on a single operation called *unification* to manipulate FDs and make decisions. Unification is increasingly used in linguistic theories (HPSG, LFG, CUG and GPSG are all recent examples of this trend cf. [Shieber 86] for a survey) and has been found appropriate for the description of many complex linguistic phenomena. In particular, there is no distinction between decision making and realization (structure construction) as is the case in NIGEL.
- *Flexible order of decision*: Unification being a bidirectional monotonic process, it is possible to make decisions in any order, and the unification mechanism propagates the decisions as in a bidirectional constraint satisfaction system. This feature makes writing grammars easier (the grammar writer does not need to think about the way a constraint interacts with other decisions) and more modular (each constraint needs to be specified only once). Flexibility in the order of decision is also important because it allows the processing of mixed input, containing both semantic and linguistic constraints. More specifically, order of decision is characterized by:
  - *Not strictly left-to-right*: In FUG, all decisions that can be made at the top-level are made before producing constituents. These decisions can send constraints down to lower levels if necessary. Thus some decisions about later sentence constituents can be made before decisions about prior constituents in the sentence. This is important when a decision made early on in the sentence depends on a decision made later.
  - *Bidirectional*: Specifying dependence of a decision on a constraint automatically specifies the inverse because of the use of unification. If the constraint is unspecified on input it will get filled in when the otherwise dependent decision is made.
  - *Interaction between different types of constraints is determined dynamically*: How different constraints interact can be determined at run-time depending on the current context of generation. This means the grammar can be modularized by constraints with specific interaction left unspecified.

[15]Cf. [Ait-Kaci 84] for the description of a programming language based on Ψ-terms, a variation of functional descriptions and [Johnson 88, pp.88-93] for a description of the encoding of arbitrary Turing machines into functional descriptions

[16]In this input, the transfer of possession is described as a composition of a material action (John does something to Mary), and a possessive relation (Mary owns the book). This analysis is derived from [Fawcett 87] and is explained in detail in Chap.5.

```
((cat clause)
 (process ((type composite)
           (relation-type possessive)
           (lex "give")))
 (participants ((agent ((cat proper) (lex "John")))
                (affected ((cat proper) (lex "Mary")))
                (possessor {^ affected})
                (possessed ((lex "book")
                            (cat common)
                            (definite no)
                            (describer ((lex "blue")))))))))
```

Figure 3-1: FUG input for *John gives a blue book to Mary*

3.1.2.2. NIGEL and PENMAN

NIGEL and the PENMAN generation system have been developed at the University of Southern California's Information Sciences Institute over the past 12 years. NIGEL is the grammatical module while PENMAN encompasses the whole generation environment, with, in addition to the surface grammar, a lexicon, an input specification language called SPL [Kasper 89], and the most general part of a knowledge base called the *upper model* [Bateman et al. 90]. NIGEL is a faithful implementation of the systemic approach to grammar presented in [Halliday 85]. Systemic linguistics has had a large influence in generation since Winograd's early work on the SHRDLU system [Winograd 72] and Davey's PROTEUS [Davey 79]. SHRDLU which was based on systemic principles demonstrated that the systemic approach offered a natural fit to many computational problems. The influence of systemic linguistics is particularly strong in the field of generation because from the beginning systemists have given priority to generation as opposed to interpretation, describing language as a set of resources to satisfy communicative needs. The process through which language serves to fulfill a certain function is called *realization*.

Systemic linguistics derives its name from the primary role that *system networks* play in the description of the realization process. A system network is ''a large directed acyclic graph whose nodes are choice points, which are called systems'' [Matthiessen 88, p.1]. An example of system is shown in Fig.3-2.

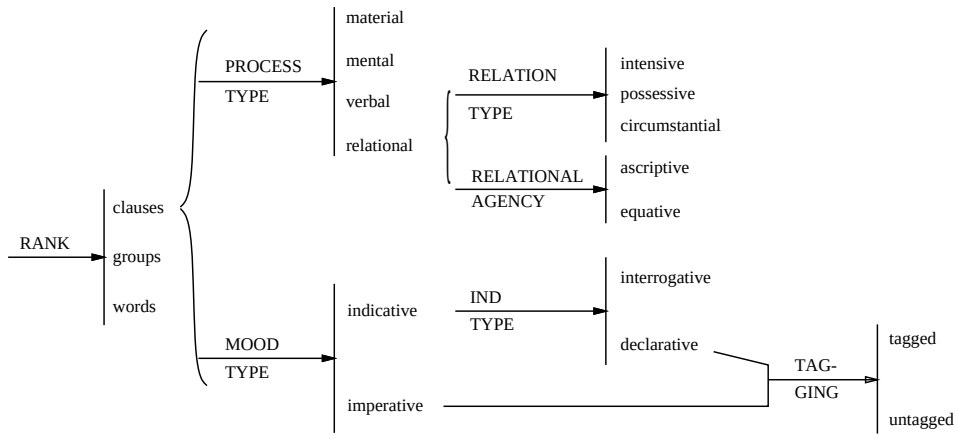


Figure 3-2: System network fragment [Matthiessen 88, p.2]

Each choice point is called a system and represents one of the decisions that must be made to produce language.

Each system has an entry condition (which is a conjunction of features) and an output feature. The system is triggered when the entry condition is satisfied, and a decision must then be made, which consists in choosing one of the output features. The effect of traversing a system is thus to set a feature[17] and as features get set, the navigation through the system network proceeds.

In addition, when they are set, features can trigger *realization statements*. Realization statements incrementally build the linguistic structure as decisions are made in the system network. For example, when the mood indicative is selected, the realization statement *insert subject* is triggered, therefore enforcing the constraint that clauses at the indicative mood have an instantiated subject. There is a language of realization statements which is briefly described in Fig.3-3.

Symbol	*Name*	*Example*
\	Realization	\ "Oh"
/	Conflation	Subject / Agent
+	Insertion	+ Subject
:	Preselection	: relative
::	Lexical preselection	:: adverb-ly
()	Expansion	(Subject)
^	Ordering	Subject ^ Verb
<>	Inclusion	

Figure 3-3: Realization statements in NIGEL

The eight realization operators used in NIGEL are listed in Fig.3-3. They define how the linguistic structure can be manipulated as decisions are made. In NIGEL, little attention is paid to the constituent structure. In fact, the NIGEL system does not support recursive constituent structures, and only finite pre-defined templates can be used. This makes the handling of conjunction, for example, particularly difficult in the grammar. A linguistic constituent is described by a set of features plus an ordered set of functions (the sub-constituents). Order is specified independently of the constituent structure through the use of the ordering realization statement (*X* ^ *Y* stands for *subconstituent X precedes Y in the realization*). The realization statements completely define how to build up such a structure.

The distinction between decision making and realization is the most important difference between NIGEL and FUGs. In FUGs, both decision making and realization are performed by the same operation of unification. A statement `(attribute val)` in the grammar can be read as: if `attribute` already has a value compatible with `val`, proceed with this branch of the grammar, else if `attribute` has no value in the current linguistic structure, enrich it with the pair `(attribute val)` and thus build up the linguistic structure, else if `attribute`'s current value is not compatible with `val`, choose another branch of the grammar. All of these cases are uniformly dealt with by the unification procedure which is expressive enough to capture all of the system network decision making formalism plus all of the realization statements.

The main contribution of NIGEL and the PENMAN system is to specify a clean interface between the grammatical system and the environment, defining a framework known as *chooser and inquiry semantics*. The generation process in PENMAN can be viewed as a traversal of NIGEL's system network. At each choice point, the entry conditions of the systems are checked. When they are satisfied, a decision must be made. This is where the environment is checked by invoking a chooser procedure. A chooser is a specialized procedure responsible for making purposeful decisions in the grammar. For example, in order to choose the mood of a clause, the grammar needs to know what is the illocutionary force of the communicative act to be realized. This force must be defined somewhere in the environment. The grammar chooses how to navigate in the mood system by calling the chooser `MOOD-TYPE-CHOOSER` (shown in Fig.3-4). This chooser makes a decision by querying the environment for specific information, through the invocation of an inquiry. Inquiry `COMMAND-Q` in this case can either access a knowledge base, an inferencing system, or query the user of the grammar (Fig.3-4).

[17]Features in systemic linguistics are either set or not (they are binary). In contrast, a feature in FUGs is a pair (attribute value), where value can cover any range of options. Systemic features are therefore the special case of FUGs features of the form (feature +/-).

```
(
The Chooser MOOD-TYPE-CHOOSER is:
Formal:
   ((ASK (COMMAND-Q SPEECHACT)
      (COMMAND (INDENTIFY SUBJECT (COMMAND-RESPONSIBLE-ID SPEECHACT))
               (CHOOSE IMPERATIVE))
      (NOCOMMAND (CHOOSE INDICATIVE))))
)


(
Inquiry: COMMAND-Q
  Domain: TP
  Mode: ENVIRONMENT
  Parameters: (ACT1)
  English:
  ("Is the illocutionary point of the surface level speech act
    represented by"
  ACT1
  " a command, i.e., a request of an action by the hearer?")
  Operatorcode: COMMANDCODE
  Parameter Association Types: (CONCEPT)
  Answerset: (COMMAND NOCOMMAND)
  Preselection Guidance: NIL
  Created Association Type: NIL
  Trivial Default: NIL
)
```

Figure 3-4: Chooser and inquiry in PENMAN [Matthiessen 88 Vol.3]

The set of choosers with their inquiries functionally define what information the environment must be able to provide in order to drive the generation process. The overall PENMAN approach is summarized in Fig.3-5, showing the central role of the grammatical network, with the chooser and inquiry interface on top of it, and the realization statements below the grammar.

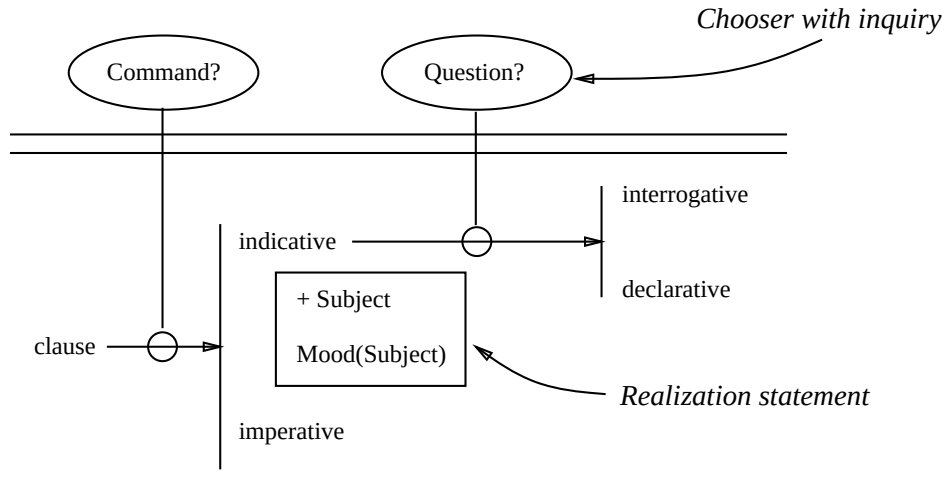


Figure 3-5: The PENMAN architecture: systems, choosers and realization

NIGEL expects an extremely rich input, which consists of the set of answers to all the queries activated during the traversal of the grammar network. Within the PENMAN system, an input specification language was designed to

allow an easier and less verbose interaction with NIGEL. The SPL interpreter allows the user to enter specifications in a much simpler way and to only partially specify the features that must be expressed. Figure 3-6 shows a possible SPL specification that might be used as input to generate *John gives a blue book to Mary*. All the features that are not specified in the SPL input are given a default value. SPL and the defaulting mechanism are described in [Kasper 89]. Note that in the SPL input, the values of the slots refer to entities in a knowledge base (the domain model). When needed, NIGEL will query this knowledge base to make a decision using the appropriate inquiry and choosers.

```
((GIVE1 / GIVE            ;; GIVE1 is an instance of GIVE
   :actor JOHN1           ;; in the domain model
   :destination MARY1
   :object BOOK1
   :tense PRESENT
   :speechact ASSERTION)
 (JOHN1 / PERSON
   :name John)
 (MARY1 / PERSON
   :name Mary)
 (BOOK1 / BOOK
   :determiner A
   :relations  ((C1 / COLORING
                  :domain BOOK1
                  :range BLUE)))
 (BLUE / COLOR))
```

Figure 3-6: Input to NIGEL for *John gives a blue book to Mary*

For a control strategy, NIGEL's network traversal is deterministic (There is no backtracking), and decisions are made in a fixed order: left-to-right traversal of the network and top-down traversal of the syntactic constituent structure. Within each constituent, decisions are made in order of ''delicateness'' (a systemic technical term defining roughly what are the decisions having the greatest impact on the linguistic realization). A more flexible approach to control in system networks is discussed in [Patten 86]. Because of the fixed order of decision making, many inquiries need to be sent to the environment early on, while processing the least delicate systems, even though a later inquiry (in a more delicate system) could have shown that the question was not necessary. For example, NIGEL always asks whether a clause should be indicative or imperative (one of the first systems in the clause network), even though later choices in the grammar make it clear that the clause must be indicative (for example, because it must be interrogative). Patten also identifies another problem with the fixed order strategy:

> Another problem NIGEL faces as a result of the chooser approach is the problem of coordinating the decisions made by the different choosers - which may be interdependent. When the traversal reaches the system to choose the number of an indicative Subject, the chooser asks the environment: *Is the [Subject] inherently multiple, i.e., a set or collection of things, or unitary?* [Mann & Matthiessen 83b]. The problem is that at this point in the traversal, the Subject may not yet be conflated with another function for which this information is available. In that case, the environment cannot answer the question. Mann suggests that it would be unreasonable to suspend the decision until later because the entire system might eventually get stalled. He therefore suggests that the grammar could be rewritten so that the choices are guaranteed to be made only when the information is known at that point of the traversal [Mann 85]. It seems unreasonable, however, to place restrictions on the grammar to suit NIGEL's computational characteristics - clearly, this defeats the purpose of using a linguistic grammar. [Patten 86, p.144]

Patten, with the SLANG system, proposes a less procedural interpretation of the system network, viewing it as a sort of rule base, that can be processed either forward or backward. He also suggests that all explicit choices at the semantic level be made before the grammar is processed. This again imposes certain restrictions on order of decision making (in particular preventing the use of mixed input), but is better motivated than NIGEL's approach to control.

Finally, NIGEL is grammar-directed, in the sense that the grammar is traversed and the input and knowledge sources are queried when decisions need to be made. In contrast, as discussed below, MUMBLE is message-directed, in the sense that the generation process traverses the input message. As noted by Meteer, this raises the problem of insuring complete coverage of the input:

> This raises the problem of knowing whether the utterance built conveys the entire message. If something unexpected is in the message, it won't break the system; it simply won't be included in the utterance. [Meteer 89, p.63]

As discussed below, FUF is also grammar-directed in the sense that the grammar is traversed, but the problem of insuring complete coverage is not as acute, since the syntactic information and the input message are expressed in the same formalism and are merged into a single output structure that must be compatible with both. It is therefore possible to impose coverage constraints in the input (cf. in particular the use of the ANY existential constraint discussed below).

In summary, NIGEL and PENMAN implement a systemic approach to text generation, giving a central role to the grammar viewed as a network of choice points. Priority is given to the functional perspective on language over the structural one. Decision making is modeled as navigation through the network and instantiation of features. Realization is distinguished from decision making and builds up the linguistic structure as the navigation proceeds. Order of decision is fixed and deterministic. Because the system network determines the order of decisions, NIGEL's approach is best viewed as a procedural grammar. Emphasis is put on defining a clean interface between the environment and the grammar through the use of choosers and inquiries.

3.1.2.3. MUMBLE

MUMBLE evolved from McDonald's original work in 1972, as a reworking of Winograd's SHRDLU, up to the MUMBLE86 implementation [Meteer et al. 87]. Instead of a complete generation system (including content planning), research in MUMBLE has focused on the realization component, encapsulating syntactic knowledge. MUMBLE takes as input a fully lexicalized compositional structure and handles the linguistic constraints on realization. Several principles have guided the development of MUMBLE:

- **Psychologically plausible model of speech production**: in MUMBLE, emphasis is on efficiency. Many of the design principles underlying MUMBLE can be related to the requirement that all decisions made by the system be indelible (no decision is ever revised) and that the production of the word stream proceeds in a strict left-to-right order.
- **Multiple levels of representation**: the stream of decisions is viewed in a modular way as acting on different levels of representations, each optimized to allow efficient implementation. Specifically, in MUMBLE, three levels are used: *realization specification* (input), *surface structure* (recording all syntactic attachments between constituents) and *word stream* (used to perform linearization and morphological operations).
- **Procedural orientation**: in MUMBLE, each level of representation is viewed as a set of instructions which, when executed by the appropriate virtual machine, produces the next level of representation. The first mapping is called *realization* in MUMBLE and transforms realization specifications into surface structures. The second mapping is called *phrase structure execution* and transforms surface structures into word streams.

From a linguistic perspective, in MUMBLE, emphasis is on the structural view of language and the constituent structure of linguistic objects. In particular, MUMBLE relies on a notion of *attachment*, close to the adjunction operation defined in TAGs.

An example of realization specification is shown in Fig.3-7 (derived from an example in [Meteer et al. 87p.90]). Note that MUMBLE requires all lexical heads to be chosen in the input and their arguments to be instantiated. For the modifiers that are not lexical arguments, the way the modifier is attached to the head must be specified syntactically, as specified by the `attachment-function` in the Fig.3-7.

Figure 3-8 shows the phrase structure to which this realization specification is mapped. The arrows indicate in which order the constituents are traversed during the phrase structure evaluation stage of the generation.

MUMBLE requires in input three elements [Meteer et al. 87, p.2]:

- The basic constituents of the utterance
- The functional relationships among the units in terms of head, predication and optional modifiers.
- Lexical choice: lexical heads are all specified.

MUMBLE is therefore not in charge of mapping a semantic structure to a non-isomorphic syntactic structure. The lexical heads with their arguments, and therefore most of the linguistic structure, are all specified in the input.

```
((discourse-unit <r1>
   :head (general-clause <r2>
           :head (give <r3>
                   (general-np <r4>
                     :head (np-proper-name "John") <r5>
                     :accessories (:number singular
                                   :determiner-policy no-determiner))
                   (general-np <r6>
                     :head (np-proper-name "Mary") <r7>
                     :accessories (:number singular
                                   :determiner-policy no-determiner))
                   (general-np <r8>
                     :head (np-common-noun "book") <r9>
                     :accessories (:number singular
                                   :determiner-policy kind)
                     :further-specifications
                       ((:specification
                         (predication_to-be *self* (adjective "blue"))
                         :attachment-function restrictive-modifier)))))
   :accessories (:tense-modal present
                 :progressive
                 :unmarked)))
```

Figure 3-7: MUMBLE's input for *John is giving a blue book to Mary*

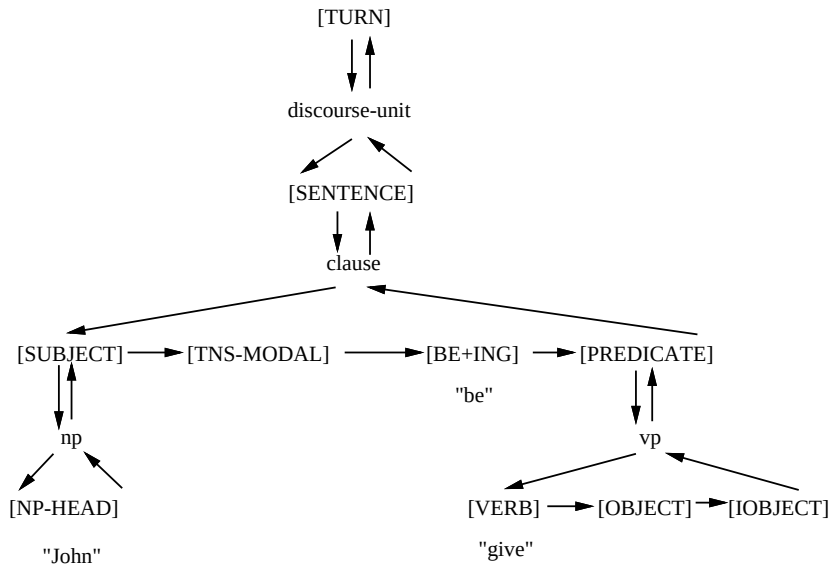


Figure 3-8: Phrase structure generated by MUMBLE (derived from [Meteer et al. 87, p.101])

The mapping from the input to the phrase structure is mainly realized by making choices in *realization classes*. The realization class is the main encoding of the decisions that the generator must make and therefore plays a role similar to systems in NIGEL and alts (disjunctions) in FUGs. MUMBLE insists that all choices be deterministic and never undone. The grammar writer must therefore pay special attention to the ordering of decisions, to avoid situations where an earlier decision leads to a dead-end. The mechanism to control order of decisions is called *bundle-driver*. Bundle drivers are procedures that control how a set of constituents is to be attached to a head constituent. The control mechanism in MUMBLE is therefore completely procedural and tightly controlled by the assumption of determinism.

Finally, as mentioned above, MUMBLE is the only formalism presented here which is message-directed: the generation process traverses the input message, not the grammar.

In summary, MUMBLE provides a modular view of the realization process, by defining a set of three representation levels. In MUMBLE, emphasis is put on efficiency and deterministic processing, to the cost of flexibility in the grammar and input specification. Finally, MUMBLE relies on a mainly procedural description of linguistic knowledge.

3.1.3. Summary: Comparison of Generation Formalisms

The following table summarizes the features of the three formalisms reviewed in this section using the criteria listed on p.43.

	FUG	NIGEL	MUMBLE
Description of conceptual input	yes	yes	Lexicalized (Note 1)
Partial input description	yes	with SPL	no
Mixed input	yes	no (Note 2)	no
Functional perspective	yes	yes	no
Structural perspective	yes	weak	strong
Declarative/procedural	declarative	(Note 3)	procedural
Order of decision making	flexible	fixed	fixed
Efficiency	Worst case exponential	(Note 4)	good (Note 5)
Reversability	Possible	no	no

Note 1: The input to MUMBLE is fully lexicalized and the assumption is that all conceptual decisions are reflected in the choice of lexical heads. So MUMBLE uses lexical items as a conceptual specification. In the comparison made here, the other formalisms also rely on a fully lexicalized input, but the input contains a conceptual description in addition to (sometimes, in place of) the lexical items.

Note 2: It seems that NIGEL does not allow linguistic features to be placed in the input, as all features are set by a chooser in response to an inquiry. The input must only therefore consist of answers to inquiries, which are semantic in nature. But the defaulting mechanism of SPL could probably be used to implement a mixed input.

Note 3: NIGEL's implementation procedural but it does include a declarative component. The current implementation is procedural because the system network is always traversed in the same order. It includes a declarative component because this same system network could be interpreted in different ways by a more sophisticated implementation. Patten's SLANG system presents one such declarative system interpreter [Patten 86].

Note 4: NIGEL's complexity and efficiency are difficult to assess. In practical terms, NIGEL seems to be much slower than FUF. I am not aware of any formal complexity analysis of the NIGEL algorithm, although, being a deterministic traversal of the system network, and given that recursive constituent structures are not supported, it should be linear in the size of the grammar (number of choice points encountered) and the depth of the most complex syntactic structure it can generate.

Note 5: Research in MUMBLE has emphasized efficiency and bounded worst-case behavior. It should be noted that in practical terms again, FUF appears to run faster than MUMBLE on the same class of inputs. FUF can become slow on inputs that neither MUMBLE nor NIGEL can handle (because they require non-deterministic search). Efficiency is discussed in detail in the next chapter.

In conclusion, FUG appears as the most flexible formalism of the three reviewed here, in terms of input specification, order of decision making and potential for reversability. The main problematic issue for FUGs is their worst case complexity. The problem of efficiency is addressed in the next chapter, with extensions in FUF increasing the efficiency of the formalism for complex input configurations.

Another dimension, not addressed in this review, is usability: how easy each formalism is to use, how readable are input and grammar specifications, how well does the formalism scale up when the grammar grows. These issues are also discussed in the case of FUF in the next chapter.

For the goal pursued in this work, of using a single uniform formalism across the whole generation process, the flexibility of FUG makes it the only potential candidate.

3.2. An Overview of Functional Unification in Generation

This section is a high-level introduction to the use of FUGs for surface generation. It serves as a tutorial for the main concepts and provides a description of the overall process. I first informally introduce the concepts of FDs (functional descriptions) and unification by walking through an example. A toy grammar is presented and the generation of a complete clause is traced. The main constructs of FUGs are thus introduced as they occur in the example.

In the rest of this chapter, I describe the FUF implementation of the abstract FUG formalism. FUF is a LISP implementation which makes certain points left unclear in Kay's high-level description more precise. In particular, the use of paths (especially relative paths) in FDs is made precise and seems to be different from Kay's usage. A control strategy is also specified for the unifier, which was not done in the abstract formalism. In addition to these precisions, FUF introduce novel extensions to the formalism. The implementation aspect is presented in this chapter. The extensions are presented in the next chapter.

The next section provides a more technical description of the FDs as used in FUF, and presents all available unification mechanisms available in FUF. This section is thus a short ''getting started'' manual, the next one can serve as a reference manual for users of the FUF package.

3.2.1. Functional Descriptions

An FD (functional description) is a data structure describing constraints on an object. It is best viewed as a list of pairs (attribute value). An attribute is always a simple label. A value can either be a simple label or, recursively, an embedded FD. A simple example is shown in Fig.3-9. FDs are often written down in an array form, as shown in Fig.3-10. In this dissertation, I use the more compact LISP notation, except when the array notation makes a point easier to understand.

Input to generate: *The system advises John.*

```
I1 =    ((cat clause)
         (tense present)
         (action   ((lex "advise")))
         (agent    ((lex "system") (proper no)))
         (affected ((lex "John"))))
```

Figure 3-9: FD I1 represents the input to a generator

In FUGs, the same formalism is used to represent both the input expressions and the grammar. A minimal grammar that contains just enough to generate the simplest complete sentences is shown in Fig.3-11. This simple grammar illustrates the basic form of a FUG grammar. The skeleton of a grammar is always the same: a top level `alt` construct. An `alt` encodes a disjunction between several branches. For example, the FD `(a ((alt (1 2))))` represents the constraint that the value of `a` must be either `1` or `2`. In the grammar, the top level disjunction includes a branch for each syntactic category: clause, NP and verb. In general, disjunctions in the grammar describe linguistic alternatives; that is, the set of decisions a generator must make to produce a sentence.

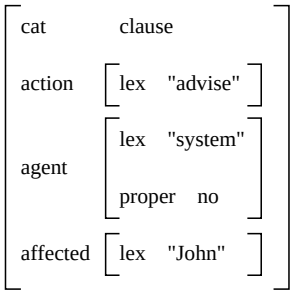


Figure 3-10: Array notation for FDs

```
((alt GSIMPLE (
   ;; a grammar always has the same form: an alternative
   ;; with one branch for each constituent category.

   ;; First branch of the alternative
   ;; Describe the category clause.
   ((cat clause)
    (agent    ((cat np)))
    (affected ((cat np)))
    (action   ((cat verb-group)
               (number {agent number})))
    (cset (action agent affected))
    (pattern  (agent action affected)))

   ;; Second branch: NP
   ((cat np)
    (head ((cat noun) (lex {^ ^ lex})))
    (number  ((alt np-number (singular plural))))
    (alt (  ;; Proper names don't need an article
           ((proper yes)
            (pattern (head)))

           ;; Common names do
           ((proper no)
            (pattern (det head))
            (det ((cat article) (lex "the")))))))

   ;; Third branch: Verb
   ((cat verb-group)
    (pattern (v))
    (aux none)
    (v ((cat verb) (lex {^ ^ lex})))))))
```

Figure 3-11: The grammar GSIMPLE

3.2.2. Unification

The only operation defined on FDs is unification. Unification is the operation that takes as input two FDs and returns as output a single FD that is compatible and more specific than both input FDs. When viewing FDs as sets of features, unification is intuitively very close to the operation of set union. For example, the FDs `((a 1))` and `((b 2))` unify into `((a 1) (b 2))`. There are two complications however: first, when the value of an attribute in the first argument is an embedded FD, the corresponding value in the second argument must be recursively unified.

For example, `((a ((x 1))))` and `((a ((y 2))))` unify into `((a ((x 1) (y 2))))`. Second, unification can fail as opposed to set union. Failure occurs when two distinct labels conflict for the value of the same attribute. For example, `((a 1))` and `((a 2))` fail to unify, because the attribute `a` can have only one value.

In FUG, generation is performed by unifying the input FD with the grammar. The input is an FD representing what must be said and the grammar is an FD representing the resources of the language to convey the input. The grammar enriches the input with the linguistic information required to produce a sentence. To unify an input FD with a disjunction, the unifier non-deterministically picks a branch that is compatible with the input and unifies the input with the selected branch. This disjunction resolution is the main mechanism through which the generator makes decisions. When unifying input `I1` with grammar `GSIMPLE`, the first decision made is to select a syntactic category. In this case, the decision is simple because the category is specified in the input with the feature `(cat clause)`. So branch 1 is selected and unified with `I1`. The first function fulfilled by branch 1 is to specify the syntactic category of each of the constituents of the clause: `agent` and `affected` are NPs and `action` is a verb. This information is added to the input.

The next main function of the grammar is to give constraints on the ordering of the words. This is done using the `pattern` special attribute. A `pattern` is followed by a picture of how the constituents of the current FD should be ordered: `(Pattern (agent action affected))` means that the `agent` constituent should come just before the `action` constituent, which should come just before `affected`.

Branch 1 also enforces the subject-verb agreement in the clause: the number of the `agent` will appear in the input as a feature of the FD appearing under `agent`, as in: `(prot ((number singular) (lex "John")))`. To enforce the subject/verb agreement, the grammar picks the feature `number` from the `agent` sub-fd and requests that it be unified with the corresponding feature of the `action` sub-fd. This is expressed by the feature:

```
(action ((number {agent number}))).
```

which means: the value of the `number` feature of `action` must be the same as the value of the `number` feature of `agent`. The curly-braces notation denotes what is called a *path* which is a pointer within an FD.

Finally, branch 1 indicates what is the linguistic structure of the clause by identifying its constituents. This is done by using the `cset` special attribute. A `cset` is followed by a list of the attributes in the FD which are constituents. In the clause, `agent`, `action` and `affected` are identified as the constituents of the clause. The `cset` feature has roughly the same meaning as writing a phrase structure rule of the form: `Clause --> Agent Action Affected`. The only difference is that in FUG, ordering constraints and immediate dominance are separated and expressed using two distinct mechanisms (`pattern` and `cset`), as is advocated by most modern grammatical theories (most notably GPSG [Gazdar et al 85] with the ID/LP format).

Thus, FDs contain both functional and structural indications. Structure is reflected by three devices: the embedding of features, the use of the special attribute `cat` which specifies the category of a constituent, and the special feature `cset` (*constituent set*) which identifies the constituents of an FD (this explains why `agent` is a constituent but `tense` is not). These three devices replace the use of phrase structure rules in other formalisms (ATNs, DCGs or even PATR-II [Shieber 86]).

The unification of `I1` with `GSIMPLE` after branch 1 is processed is shown in Fig.3-12. I1 has been enriched by the features shown in uppercase in the figure. However, the sub-fds `agent`, `affected` and `action` have not yet been unified with the grammar. So for example, after the top level unification, the input does not specify whether the `agent` is a proper noun or not. This is why the unification procedure includes a second stage. The constituents of the top level FD are first identified, and then, each sub-constituent is unified again with the whole grammar. In this example, the constituents are `action`, `agent` and `affected` (as indicated by the `cset`). They are unified recursively with the branches of the grammar for `verb`, `np` and `np` again. The final result of the unification after this recursive traversal of the constituent structure is shown in Fig.3-13. Thus the general unification procedure includes the following steps:

1. Unify top level input with grammar.
2. Identify constituent set in resulting FD.

```
((cat clause)
 (tense present)
 (action   ((lex "advise")
            (CAT VERB)
            (NUMBER {AGENT NUMBER})))
 (agent    ((lex "system") (proper no)
            (CAT NP)))
 (affected ((lex "John")
            (CAT NP)))
 (CSET (ACTION AGENT AFFECTED))
 (PATTERN (AGENT ACTION AFFECTED)))
```

Figure 3-12: Top level unification of I1 with GSIMPLE

3. Recursively unify each constituent with the grammar.

```
((cat clause)
 (tense present)
 (action   ((lex "advise")
            (cat verb-group)
            (number {agent number})
            (PATTERN (V))
            (AUX NONE)
            (V ((CAT VERB) (LEX {^ ^ LEX})))))
 (agent    ((lex "system") (proper no)
            (cat np)
            (HEAD ((CAT NOUN) (LEX {^ ^ LEX})))
            (NUMBER SINGULAR)
            (PATTERN (DET HEAD))
            (DET ((CAT ARTICLE) (LEX "the")))))
 (affected ((lex "John")
            (cat np)
            (HEAD ((CAT NOUN) (LEX {^ ^ LEX})))
            (NUMBER SINGULAR)
            (PROPER YES)
            (CSET (HEAD))
            (PATTERN (HEAD))))
 (cset (action agent affected))
 (pattern (agent action affected)))
```

Figure 3-13: Complete unification of I1 with GSIMPLE

A few more FUF constructs are introduced in branches 2 and 3 of the grammar, which deal with the NPs and the verb-group. In the NP branch, a sub-fd called `head` is first constructed by the grammar. `Head` is of category `noun` and has a feature `lex`. By convention, the feature `lex` has always for value the string representing the words to be used in the sentence. The value of `lex` in the grammar is `{^ ^ lex}`. This is a path expression, similar to the one used in branch 1 to enforce subject/verb agreement. The difference here is that the path starts with the special symbol `^`. Such a path is called a *relative path*, and points to a feature which is above the current feature in the structure of the FD. This notation is very close to the LFG up and down arrow notation. The details are explained below, and for now it is sufficient to read such a path as stating that the `lex` of the `head` feature is the same one as the `lex` of the `agent` constituent.

Next, the grammar specifies the number of the NP. Note that the `alt` disjunction can be embedded at any level of the grammar. The grammar constrains that the number be either `singular` or `plural`. Since no other constraint is given here, the unifier simply picks the first branch of the alt, and enriches the input with `(number singular)`. The ordering of the branches in an `alt` thus imposes a default choice in the grammar.

The second choice in the NP branch is between proper and common nouns. Proper nouns have no article. Common nouns do, and in `GSIMPLE` the article is always *the*. In each case, a different `pattern` determines what are the sub-constituents of the NP. Again, the order of the branches determines that proper nouns are the default, and therefore in order to produce a common noun, the feature `(proper no)` must be added to the input to override this default.

The `verb-group` branch does not use any new construct. In this toy grammar, it does not allow for any auxiliaries, modals or negation. This branch therefore does not contain any alt.

The operation of the unifier can be traced, and an inspection of the trace provides a good understanding of the unification procedure. Figure 3-14 shows the trace of the unification of `I1` with `GSIMPLE`. In the figure, each step of the unification can be identified: first the top level category is identified: `(cat clause)`. The input is unified with the corresponding branch of the grammar (branch #1). Then the constituents are identified. There are here 3 constituents: `action` of cat `verb-group`, `agent` of cat `np` and `affected` of cat `np`. Each constituent is unified in turn. Then for each constituent, the unifier identifies the sub-constituents. In this case, no constituent has a sub-constituent, and unification succeeds. In general, the hierarchy of constituents is traversed breadth first.

The example shown in Fig.3-15 illustrates how unification can fail. The input FD `I2` tries to override subject/verb agreement, causing the failure.

3.2.3. Linearization and Morphology

Once unification has succeeded, the unified FD is sent to the linearizer, to produce a string. The linearizer works by following the directives included in the `pattern`. The linearization procedure includes the following steps:

1. Identify the `pattern` feature in the top level: for `I1`, it is `(pattern (agent action affected))`.
2. If a pattern is found:
    a. For each constituent of the pattern, recursively linearize the constituent. (That means linearize `agent`, `action` and `affected`).
    b. The linearization of the FD is the concatenation of the linearizations of the constituents in the order prescribed by the pattern feature.
3. If no pattern is found:
    a. Find the `lex` feature of the FD, and depending on the category of the constituent, the morphological features needed. For example, if the FD is of `(cat verb)`, the features needed are: `person, number, tense`.
    b. Send the lexical item and the appropriate morphological features to the morphology module. The linearization of the fd is the resulting string. For example, if `(lex="advise")` and the features are the default values (as they are in `I1`), the result is *advises.* When the FD does not contain a morphological feature, the morphology module provides reasonable defaults.

Note that if a pattern contains a reference to a constituent and that constituent does not exist, nothing happens: the linearization of an empty constituent is the empty string. Finally, note that if one of the constituent sent to the morphology is not a known morphological category, the morphology module can not perform the necessary agreements, and returns the lex string unchanged.

Only the leaves of the constituent structure are sent to the morphology module. In the FUF implementation, the coverage of the morphology module is shown in Fig.3-16. The morphology processing can be performed in the grammar itself. The FUF morphology component is just added for the convenience of the grammar writer to avoid duplicating the same information in all grammars, and make grammars more concise.

```
LISP> (uni i1 gsimple)
-->
>STARTING CAT CLAUSE AT LEVEL {}

-->Entering alt GSIMPLE -- Jump indexed to branch #1: CLAUSE
   matches input CLAUSE
-->Updating (CAT NIL) with NP at level {AGENT CAT}
-->Updating (CAT NIL) with NP at level {AFFECTED CAT}
-->Updating (CAT NIL) with VERB-GROUP at level {ACTION CAT}
-->Enriching input with (NUMBER {AGENT NUMBER}) at level {ACTION}
-->Enriching input with (PATTERN (AGENT ACTION AFFECTED)) at level {}
-->Success with branch #1 CLAUSE in alt GSIMPLE

>STARTING CAT VERB-GROUP AT LEVEL {ACTION}

-->Entering alt GSIMPLE -- Jump indexed to branch #3: VERB-GROUP
   matches input VERB-GROUP
-->Enriching input with (PATTERN (V)) at level {ACTION}
-->Updating (CAT NIL) with VERB at level {ACTION V CAT}
-->Success with branch #3 VERB-GROUP in alt GSIMPLE

>STARTING CAT NP AT LEVEL {AGENT}

-->Entering alt GSIMPLE -- Jump indexed to branch #2: NP
   matches input NP
-->Updating (CAT NIL) with NOUN at level {AGENT HEAD CAT}
---->Entering alt NP-NUMBER
---->Updating NIL with SINGULAR at level {AGENT NUMBER}
---->Success with branch #1 in alt NP-NUMBER
-->Enriching input with (PATTERN (DET HEAD)) at level {AGENT}
-->Enriching input with (CAT ARTICLE) at level {AGENT DET}
-->Enriching input with (LEX "THE") at level {AGENT DET}
-->Success with branch #2 NP in alt GSIMPLE

>STARTING CAT NP AT LEVEL {AFFECTED}

-->Entering alt GSIMPLE -- Jump indexed to branch #2: NP
   matches input NP
-->Updating (CAT NIL) with NOUN at level {AFFECTED HEAD CAT}
---->Entering alt NP-NUMBER
---->Updating NIL with SINGULAR at level {AFFECTED NUMBER}
---->Success with branch #1 in alt NP-NUMBER
-->Updating (PROPER NIL) with YES at level {AFFECTED PROPER}
-->Enriching input with (PATTERN (HEAD)) at level {AFFECTED}
-->Success with branch #2 NP in alt GSIMPLE


[Used 5 backtracking points - 0 wrong branches - 0 undos]
The system advises John.
```

Figure 3-14: Trace of the unification of I1 with GSIMPLE

```
(setq I2 '((cat clause)
           (agent   ((lex "John") (number singular)))
           (action  ((lex "advise") (number plural)))
           (goal    ((lex "Mary")))))

LISP> (uni i2 gsimple)

>STARTING CAT CLAUSE AT LEVEL {}

-->Entering alt GSIMPLE -- Jump indexed to branch #1: CLAUSE
   matches input CLAUSE
-->Updating (CAT NIL) with NP at level {AGENT CAT}
-->Updating (CAT NIL) with NP at level {AFFECTED CAT}
-->Updating (CAT NIL) with VP at level {ACTION CAT}
-->Fail in trying PLURAL with SINGULAR at level {ACTION NUMBER}

<fail>
```

Figure 3-15: Failure of unification

```
CAT VERB:
     ENDING: {ROOT, INFINITIVE, PAST-PARTICIPLE, PRESENT-PARTICIPLE}
     NUMBER: {SINGULAR, PLURAL}
     PERSON: {FIRST, SECOND, THIRD}
     TENSE : {PRESENT, PAST}

CAT NOUN:
     NUMBER:       {SINGULAR, PLURAL}
     POSSESSIVE:   {YES, NO}
     A-AN:         {AN, CONSONANT}

CAT PRONOUN:
     PRONOUN-TYPE: {PERSONAL, DEMONSTRATIVE, QUESTION, QUANTIFIED}
     CASE:         {SUBJECTIVE, POSSESSIVE, OBJECTIVE, REFLEXIVE}
     GENDER:       {MASCULINE, FEMININE, NEUTER}
     PERSON:       {FIRST, SECOND, THIRD}
     NUMBER:       {SINGULAR, PLURAL}
     DISTANCE:     {NEAR, FAR}

CAT ARTICLE:
     NUMBER:       {SINGULAR, PLURAL}
     DEFINITE:     {YES, NO}
     A-AN of following word.

PUNCTUATION:
     BEFORE: {";", ",", ":", "(", ")", ...}
     AFTER : {";", ",", ":", "(", ")", ...}

ORDINAL, CARDINAL:
     VALUE:  a number
     DIGIT:  {YES, NO}
```

Figure 3-16: Coverage of the morphology component

3.3. Technical Description of FUF

I now proceed to a more precise description of the FUG formalism as implemented in FUF. This section is intended as a reference manual for users of the FUF package. It provides a detailed description of all the basic constructs implemented in FUF. Advanced constructs are discussed in the next chapter, which builds upon the description provided here. The descriptions provided here are intended to allow the reader to unambiguously interpret grammars written in FUF.

The following constructs are presented: first FDs are defined as partial descriptions of arbitrary objects. Then, special FDs, called meta-FDs are introduced, which refer to the status of features rather than to actual values. Disjunctions, the main representation of choice points in grammars, are then described. Then additional features of FDs are introduced: paths, which allow values in an FD to be pointers to other values, and a more general form of FDs as sets of equations between paths is described. An alternative view of FDs as graphs is provided which allows a better understanding of the function of paths. To conclude the description of FDs, a comparison with PROLOG's first-order terms is provided. The notion of constituent is then introduced, with the `cset` construct, and the structural aspects of the formalism are described. Ordering constraints on linguistic objects are then presented, with the `pattern` construct. Finally, the unification operation is formally described, with a pseudo-code definition of the main functions used in the unification algorithm.

3.3.1. FDs and Features

An FD describes a set of objects (most often linguistic entities) that satisfy certain properties. It is represented by a set of pairs `(a v)`, called features, where `a` is an attribute (the name of the property) and `v` is a value, either an atomic symbol or recursively an FD. An attribute `a` is allowed to appear at most once in a given FD F, so that the phrase *''the* `a` *of F''* is always non ambiguous.

A given attribute in an FD must have at most *one* value. Therefore, the FD `((size 1) (size 2))` is illegal. In fact FDs can be viewed as a conjunction of constraints on the description of an object: for an object to be described by `((size 1) (size 2))` it would need to have its property `size` to have both the values 1 and 2. Conversely, if the attribute `size` does not appear in the FD, its value is not constrained and it can be anything. The FD `nil` (empty list of pairs) thus represents all objects in the world. The pair `(att nil)` expresses the constraint that the value of `att` can be anything. It is therefore useless, and the FD `((att1 nil) (att2 val2))` is exactly equivalent to the FD `((att2 val2))`.

Thus, a crucial feature of FDs as a description of linguistic objects is that they are *partial* structures (cf. [Johnson 88, pp.65-74] and [Pereira 87]). When a feature is not present in the FD, it is assumed to be unknown. Therefore an FD can be used to represent the partial information, or approximation, that is known of a linguistic object. As more features get added, the approximation becomes more accurate. It is therefore possible to compare FDs in terms of their information content: the more features, the more information. Conversely, the more features, the less objects the FD denotes.

3.3.2. Meta-FDs

FUGs define two *meta-FDs*, NONE and ANY, to refer to the status of a feature in a description rather than to its value. `(label NONE)` indicates that `label` cannot have a ground value in the FD resulting from the unification. `(label ANY)` indicates that `label` must have a ground value in the resulting FD.

ANY is a very powerful constraint in that it triggers a delayed check. The way it is processed by the unifier is as follows: if when the constraint `(att any)` is processed, `att` already has a ground value, then the `any` constraint is satisfied. Otherwise, the check is delayed until the end of the unification process. When all other constraints have been processed, the `any` constraint is checked again. If `att` has a ground value then the constraint succeeds, otherwise it fails, and backtracking is triggered. The uses of the `any` constraint will be discussed in Chap.4.

Because of the delayed processing of the ANY constraints, the unification process must be separated into two stages:

1. **Unification**: all regular unification is performed. Undetermined constraints (like ANY) are delayed.
2. **Determination**: all delayed constraints are checked.

In the next chapter, the function of the determination stage is extended to handle other kinds of delayed constraints, introduced by the `wait` construct.

3.3.3. Disjunction

The main tool to represent choice points in the grammar is the disjunction. FDs in general are interpreted as a conjunction of constraints. It is also possible to define disjunctions, using the `alt` keyword. The syntax is: `(alt (d1 ... dn))`, which stands for an FD whose value is the disjunction of the `di` terms. Each term `di` is an FD called a *branch* of the alt. Figure 3-17 lists examples of disjunction in different contexts.

Disjunction of atomic values:

```
((size ((alt (1 2 3)))))
```

Disjunction of FDs:

```
((alt (((cat clause)
        (semantic event))
       ((cat np)
        (semantic object))
       ((cat adjective)
        (semantic property)))))
```

Embedded disjunctions:

```
((alt (((cat clause)
        (mood ((alt (finite non-finite))))
        (alt (((tense present)
               (mood finite))
              ((tense past)))))
       ((cat np)
        (mood none)))))
```

Figure 3-17: Examples of disjunctions

Disjunctions are processed by the unifier as follows: to unify an FD with a disjunction, non-deterministically select a branch in the disjunction. Proceed with unification. If the unification process fails (because a contradiction is found), backtrack to one of the previous choices. The selection of a branch in an alt is the only decision making mechanism in FUG.

Embedded disjunctions can always be rewritten into top-level disjunctions, in a form called *normal form* [Kay 85]. This transformation works as indicated in Fig.3-18.

The number of branches in the top level alt of an FD in normal form is exponential in the depth of the embedding of alts in the original FD. In practice, for large grammars, the normal form can contain up to 10^{29} branches.

In certain implementations, disjunctions are expanded and kept in extensive form instead of being ‘‘resolved’’ as in FUF, by choosing one branch out of the set of branches. In such an approach, the unification of an FD with a disjunction yields a new disjunction:

```
unify(FD, ((alt (d1...dn)))) = ((alt (unify FD d1)...(unify FD dn)))
```

If one of the branches `di` is not compatible with `FD`, the branch is canceled, by virtue of the equivalences:

```
((alt (d1 fail))) = d1
((alt ())) = fail
```

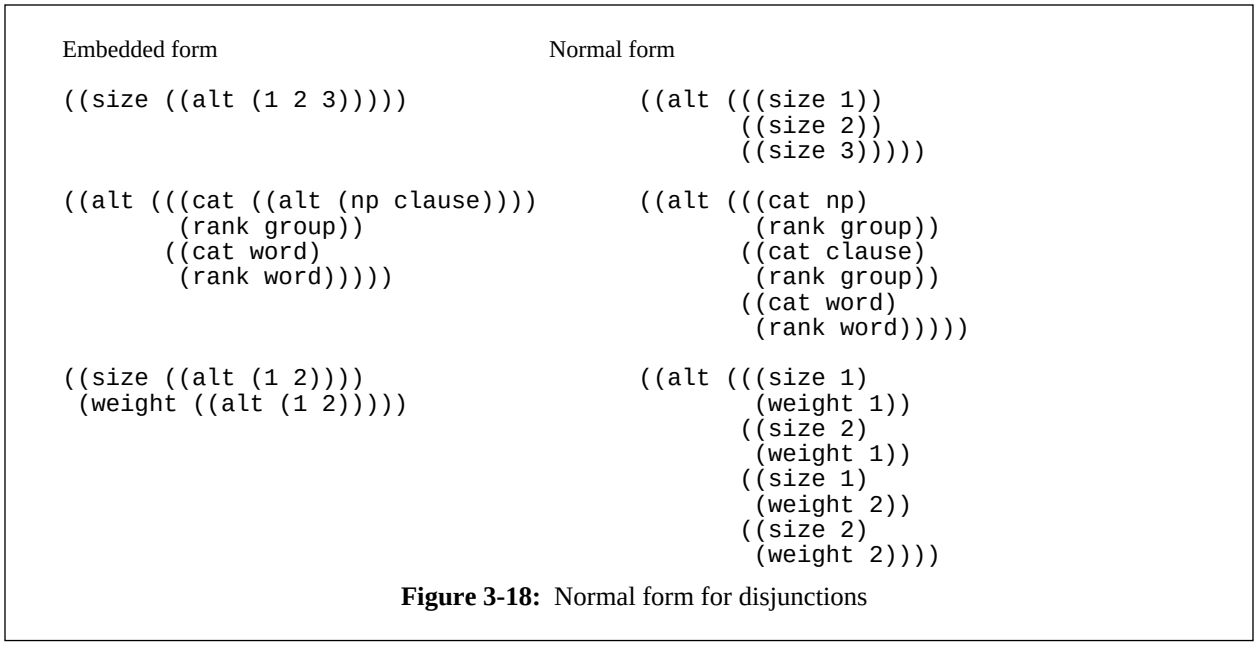


Figure 3-18: Normal form for disjunctions

where `fail` is the symbolic representation of an incompatible FD. The problem with such an approach is that in practice many disjuncts stay ''alive'' during the unification process, and the result tends to take on sizes close to the normal form expansion of the grammar, which is clearly unmanageable. [St-Dizier 92] reports on experiments using this expansion technique in a different framework. The FUF implementation does not follow this approach, although one form of the `wait` construct presented in the next chapter, allows the grammar writer to selectively keep certain disjunctions unresolved when desired.

The idiom `(alt (FD nil))` is a useful form of disjunction. It can be read as: Try to unify FD, if it succeeds keep it, otherwise, do not fail (because unification with `nil` always succeeds). So the construct can be read as `FD` is an optional constraint, and is useful in implementing a sort of default mechanism. There is in FUF a special construct `(opt FD)` which is equivalent to this form.

In FUF, the default control mechanism is to process disjunctions in the order specified in the grammar. Therefore, the grammar always picks the first untried branch of an alt, and upon failure, tries the following branch. Backtracking is purely chronological by default (upon failure, the latest choice point is revisited). More sophisticated control mechanisms to constrain the selection operation are presented in Chap.4.

3.3.4. Paths

Any position in an FD can be unambiguously referred to by the *path* leading from the top-level of the FD to the value considered. For example, FD `I1` is shown in Fig.3-19 both in its embedded form and as a list of path expressions.

Paths are represented as simple lists of labels between curly braces (for example, `{agent proper}`). This notation is not ambiguous because at each level there is at most one feature with a given attribute.

In general, I consider the unification of two FDs that I call input and grammar. Define L as a set of labels or attribute names and C as a set of constants, or simple atomic values. A string of labels (that is an element of L^*) is called a path, and is noted $\{l_1...l_n\}$. A grammar defines a domain of admissible paths, $\Delta \subset L^*$. Δ defines the skeleton of well-formed FDs. The meta-FD NONE is best viewed as imposing constraints on the definition of Δ: an equation $\{l_1...l_n\}$=NONE means that $\{l_1...l_n\} \notin \Delta$.

```
I1 =    ((cat clause)
         (tense present)
         (action   ((lex "advise")))
         (agent    ((lex "system") (proper no)))
         (affected ((lex "John"))))


{cat} = clause
{tense} = present
{action lex} = "advise"
{agent lex} = "system"
{agent proper} = no
{affected lex} = "John"
```

Figure 3-19: FD in embedded and flat forms

A path can be *absolute* or *relative*. An absolute path specifies the way from the top-level of the FD down to a value. A relative path is a pointer that starts from the spot where it occurs and can go up the FD structure along the embedding attributes. A relative paths starts with the symbol "^" (up-arrow). It refers to the FD embedding the current feature. For example, if the relative path `{^ x}` occurs at the spot located at the path `{a b c}`, as in `((a ((b ((c {^ x})))))))`, it means that the value of the path `{a b c}` is the same as the value of the path `{a b x}`. This new path is obtained by going up one level from `{a b c}`, yielding the path `{a b}`, and then down one level along the `x` attribute to `{a b x}`.

Several "^" in a row can be used to go up several levels. The notation `^n` can be used as an abbreviation of `n` consecutive up-arrows. For example, the notation `{^4 x}` is equivalent to `{^ ^ ^ ^ x}`. It is convenient when dealing with deeply embedded constituents.

Relative paths are not simply a syntactic convenience, but they extend the expressibility of the formalism, by making grammars ‘‘relocatable’’. For example, the grammar for NPs can be unified with a subconstituent of the input FD at different levels (`{agent}` and `{affected}` for example). In each case, a feature like `(determiner ((number {^ ^ number})))` points to the number of the appropriate constituent. Without relative paths such a general constraint could not be expressed.

The value of a pair can be a path. In that case, it means that the values of the pair pointed to by the path and the value of the current pair must always be the same. In this case, the two features are said to be unified. In the example discussed in Sect.3.2, subject/verb agreement was enforced by unifying the features at paths `{action number}` and `{agent number}`. The effect of this unification is that the two locations identified by these paths become identical: the two paths are two names for the same object (structure sharing). This is equivalent to the systemic operation of *conflation*, with the difference that it is a bi-directional operation: if both objects are instantiated, the unification operation checks that they are compatible (and therefore can be unified); if one of the objects is not (or only partially) instantiated, then the unification operation enforces the conflation operation by setting its value. In contrast, in the systemic framework (as implemented for example in NIGEL), conflation is uniquely a realization operation; that is, it only works as a unidirectional assignment.

The only case where a given attribute can appear in several pairs is when it is followed by paths in all but one pair. This is illustrated in Fig.3-20.

There are no restrictions in the formalism on where paths and relative paths can point, making the description of arbitrary complex graphs possible with FDs. This unrestricted layout is the source of the computational complexity of FUGs. Note, however, that in practice, a grammar can impose constraints on accessibility within a constituent structure.

```
((a ((a1 v1)))
 (a {b})
 (a {c}))
```

is a valid FD. It is equivalent for example to:

```
((b ((a1 v1)))
 (a {b})
 (c {b}))
```

or to:

```
((b ((a1 v1)))
 ({a} {b})
 (c {a}))
```

Figure 3-20: Several equivalent FDs using paths

3.3.5. Equations and Constraints

In general, an expression of the form `x = y`, where either `x` or `y` is a path or a leaf is called an equation. As mentioned above, it is possible to represent an FD as a flat list of equations. Figure 3-21 shows the syntax used in FUF. Note that FUF accepts either labels or paths as the left-hand side of a feature.

```
(({cat} clause)
 ({tense} present)
 ({action lex} "advise")
 ({agent lex} "system")
 ({agent proper} no)
 ({affected lex} "John"))
```

Figure 3-21: FUF syntax to encode FD I1 as a flat list of equations

This notation allows to freely mix the ‘‘FDs as equations’’ view with the ‘‘FDs as structure’’ one.[18] This dual representation is one of the most attractive characteristics of functional unification. In linguistics, the possibility of using a non-structured representation removes the emphasis that has traditionally been placed on structure and constituency in language.

To appreciate the benefit of this dual interpretation of FDs, consider first the example of ATNs [Woods 70]. ATNs put the emphasis on structure The network representation only encodes structural relations between linguistic constituents of different categories. A completely distinct mechanism, that of registers, is used to encode non-structural information. Two distinct formalisms effectively must be used to deal with the structural perspective and the functional one. Consider next the other extreme example of systemic grammars, which put the emphasis on a functional description. In systemic grammars, only simple features are used to represent linguistic objects. These features are of the form `(attribute +|-)`. This mechanism is obviously insufficient to represent the syntactic structure of a linguistic object, because it does not specify if a feature applies to a whole clause for example or only to one of its NPs. So again a different convention is used to encode the syntactic structure: flat lists of features are used for each constituent, and the constituent structure is often represented using a context free formalism, as explained for example in [Houghton 86]. In contrast, in FUGs, both aspects, functional and structural, are handled simultaneously by relying on this dual interpretation of FDs.

[18] Note that the possibility to put paths on the left of a pair increases the expressive power of the `external` construct discussed in Chap.4, as it becomes possible to express at unification-time constraints on constituents which are not dominated by the position of the external construct in the structure.

3.3.6. FDs as Graphs

When the structure of an FD becomes complex, and more conflations with paths are introduced, a visual representation of the FD becomes extremely useful. This visual representation also provides a clear interpretation of the path mechanism and makes reading of relative path much easier. The structured format of FDs can be viewed as equivalent to a directed graph with labeled arcs as pointed out in [Karttunen 84]. The correspondence is established as follows: an FD is a node, each pair `(attr value)` is a labeled arc leaving this node. The `attr` of the pair is the label of the arc, the value is the adjacent node. Internal nodes in the graph have therefore no label whereas leaves are atomic values. The equivalence is illustrated in Fig.3-22.

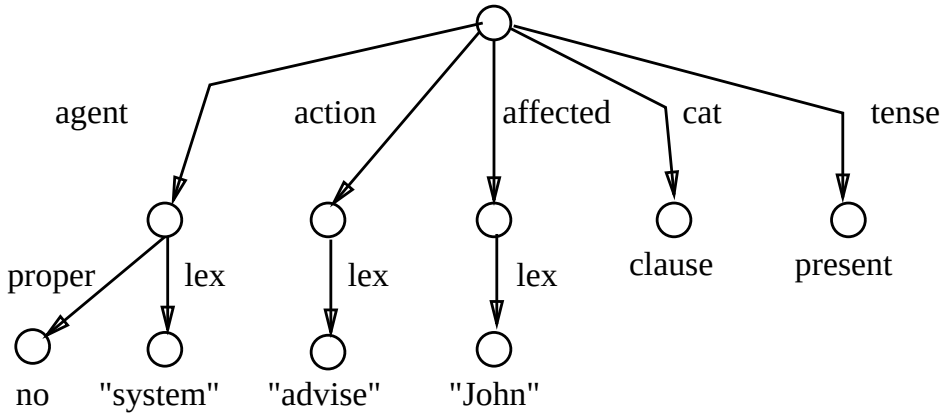


Figure 3-22: FD as a graph

The graph notation is particularly useful to interpret relative paths. When a relative path occurs somewhere in an FD, its destination can be identified by going up on the arcs, one arc for each "^". When the value of a pair is a path, e.g., `(a {b})`, then the corresponding arc actually points to the same node as the given path. In this case, there is structure sharing between a and b. This configuration is illustrated in Fig.3-23, where the paths `{action number}` and `{agent number}` are conflated, as well as the paths `{affected lex}` and `{affected head lex}` and `{subject}` and `{agent}`.

The conflation of `{subject}` with `{agent}` makes all the paths that are extensions of either `agent` or `subject` equivalent. For example, `{agent lex}` and `{subject lex}` are equivalent. This equivalence is easily read in the graph notation.

The graph notation also makes it clear that the up-arrow notation can be ambiguous. Whenever a Y configuration is met in the graph, for example in the two black nodes in Fig.3-23, the up-arrow does not specify which branch of the Y must be taken. This problem is illustrated in the grammar in Fig.3-24. The FD is extracted from a grammar dealing with conjunction. The constraint enforced by the grammar is that all the conjuncts in a conjunction must have the same syntactic category. A conjunction is represented by an FD with two constituents: `head` represents the conjunction as a whole as a constituent and `list` is a list of conjuncts. The list is represented in a singly-linked list of elements, with a recursive FD containing at each level the first element of the list (feature `car`) and the rest of the list (feature `cdr`). In Fig.3-24, the path `c1` is used to point to the first constituent of the list. `c1` is therefore defined by the equation `{c1} = {list car}`. The grammar in LISP notation is shown in Fig.3-25 along with a sample input.

The line underlined in Fig.3-25 corresponds to the black dot in the graph notation shown in Fig.3-24. The problem is to interpret where the relative path `{^ ^ head cat}` is pointing to. The notation is ambiguous between `{head`

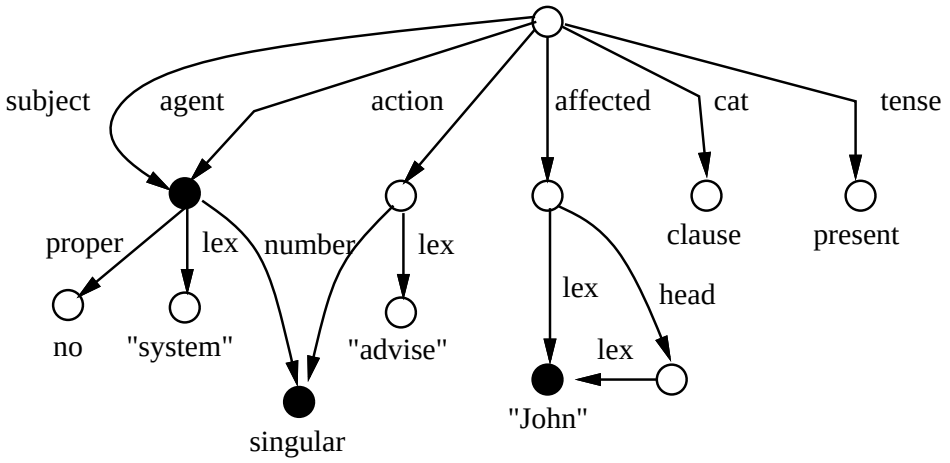


Figure 3-23: Conflation in an FD graph

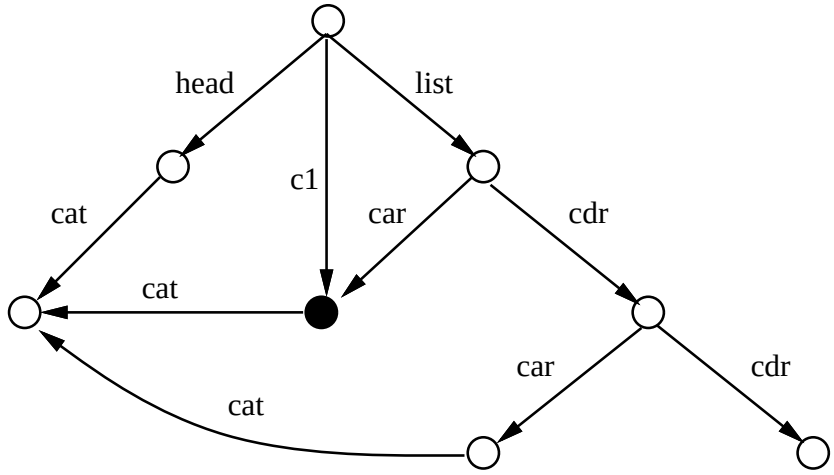


Figure 3-24: A grammar for conjunction

```
GR =    ((c1 {^ list car})
         (c1 ((cat {^ ^ head cat}))))

IN =    ((head ((cat np)))
         (list ((car ((lex "cat")))
                (cdr ((car ((lex "dog")))
                       (cdr none))))))
```

Figure 3-25: Ambiguity of the up-arrow notation

`cat}` and `{list head cat}`, depending on whether one considers the black dot as being located at address `{c1}` or `{list car}`. This ambiguity is solved in FUF by following the convention that up-arrows always refer to the textual location where they appear in the grammar. So in Fig.3-25, the up-arrows refer to the address `{c1 cat}`

and not to the address `{list car cat}` because they are written as a pair `(c1 ((cat {^ ^ head cat})))` and not as `(list ((car ((cat {^ ^ head cat}))))))`.

3.3.7. Functional Descriptions vs. First-order Terms

To conclude the characterization of FDs as a data-structure, it is useful to contrast functional unification (FU) with the more well known structural unification (SU) as used in PROLOG for example, and to distinguish FDs from the first-order terms used in SU.

The most important difference is that functional unification is not based on order and length. Therefore, `{a:1, b:2}` and `{b:2, a:1}` are equivalent in FU but not in SU, and `{a:1}` and `{b:2, a:1}` are compatible in FU but not in SU (FDs have no fixed arity). The following quote from Knight summarizes the distinction between feature structures and the first order terms used in SU:

- Substructures are labeled symbolically, not inferred by argument position.
- Fixed arity is not required.
- The distinction between function and argument is removed.
- Variables and coreference are treated separately. [Knight 89, p.105]

A comparison between the FD notation and the first-order term notation illustrates these differences. The following FD and first-order term can be used to represent the fact that *Steve builds a crane that is 2 lbs and 4 feet high*:

```
((process build)
 (agent Steve)
 (object ((concept crane)
          (weight 2)
          (height 4))))

build(Steve, Crane(C1, 2, 4))[19]
```

Contrasting these two notations for the same example illustrates the differences:

- *Symbolic labels for substructures*: the arguments, that is the `agent` and the `medium`, are clearly labeled in the feature structure notation. In particular, a term like `Crane(C1, 2, 4)` is particularly difficult for a human to interpret.
- *Fixed arity*: features can be added at will to an FD. FDs are used to represent *partial information*. This is not the case for first-order terms. If the knowledge representation changes to include width to crane descriptions, in addition to weight and height, all the terms need to be updated, since `Crane(n, w, h)` is not compatible with `Crane(n, w, h, l)`. The FD notation is always partial and leaves the possibility of adding new features as needed.
- *Function and argument*: first-order terms have a head (the function) which plays a central role in the unification process. This is not the case in FDs. All information plays the same role.[20]
- *Variables and coreference*: in standard unification, a variable is used to mean two distinct things: that the value of the role is unknown (there are no constraints on it), and that the value of the role is the same as all other objects referred to with the same variable. So for example, in a term such as `like(X, X)`, the use of the variable `X` means that we don't know who likes whom, and that it is known that the agent and the object of `like` must be the same objects. The distinction between these two functions of variables is best explained in [Ait-Kaci 84]. In FDs, coreference and unspecification are represented by

[19]Other representations are of course possible using first-order notation. Some of them have some of the advantages of features structures. For example: build(B1), agent(B1, Steve), object(B1, C1), crane(C1), weight(C1, 2), height(C1, 4). In fact, any FD can always be translated in a one-to-one mapping to a class of restricted first-order terms.

[20]In most cases, however, one feature plays a special role: for example, the `cat` attribute can specify the category or type of a description, but this is not built into the syntax, and several such ''special'' attributes can coexist in the same FD, allowing a reader to adopt several perspectives on the same FD.

two different syntactic devices: variables are features which are unspecified (they simply do not appear in the FD, or appear with an empty value), coreference is handled with *paths*. For example, to express the constraint that the medium of `like` must refer to the same object as its agent, the following FD can be used:

```
((process like)
 (agent {object}))
```

3.3.8. Constituent Unification

Two special labels are defined in the basic FUG formalism: `cset` and `pattern`. These labels are special because their value is not an FD and the procedure to unify `cset` and `pattern` is a different unification procedure than the one used for regular FDs. `Cset` is used to specify the constituent structure of an FD and `pattern` is used to express ordering constraints on the linguistic object represented by an FD. `Pattern` is further discussed below in Sect.3.3.9.

A *constituent* of a complex FD is a distinguished subset of features. The special label CSET (Constituent Set) is used to identify constituents. The value of CSET is a list of paths leading to all the constituents of the FD. Constituents trigger recursion in the FU algorithm and the `cset` attribute significantly increases the expressiveness of FUGs.

The general control mechanism in FUF is the following:

- Unify at the top level the input with the grammar.
- Identify the cset of the resulting FD.
- Recursively unify each constituent.

The constituent structure is traversed breadth-first, to make sure that least delicate decisions are made before the more delicate ones, and at each level, the order is specified by the order indicated in the `csets`. `Cset` is the primary tool through which the linguistic structure is described.

3.3.9. Ordering Constraints

A pattern placed in a constituent constrains the order in which its subconstituents can appear in the linearized string. The value of the `pattern` attribute is expressed in a special language expressing partial ordering constraints. In general, a pattern specification is made up of a list of paths and of the special symbol `dots`. `Dots` can stand for zero or more constituents. Examples of pattern specification are given in Fig.3-26.

```
((subject ...)
 (object ...)
 (verb ...)
 (pattern (dots {^ subject} dots {^ verb} {^ object} dots)))
```

Anything then subject, then anything then verb immediately followed by object then anything.

```
((pattern (dots {^ adverb} dots {^ verb})))
```

Anything then adverb, then anything then verb at the end (nothing can follow verb).

Figure 3-26: Examples of pattern specifications

A special pattern unification procedure can merge several pattern specifications into a single one compatible with all of them. The example in Fig.3-27 illustrates the fact that pattern unification is non-deterministic in general.

Patterns are eventually interpreted by the linearization component to produce a string out of an FD.

```
Pattern Unification:
p1: (pattern (dots a dots b dots))
p2: (pattern (dots c dots d dots))

Compatible Results:
(pattern (dots a dots b dots c dots d dots))
(pattern (dots a dots c dots b dots d dots))
(pattern (dots a dots c dots d dots b dots))
(pattern (dots c dots a dots b dots d dots))
(pattern (dots c dots a dots d dots b dots))
(pattern (dots c dots d dots a dots b dots))

Pattern Unification:
p3: (pattern (dots a dots b))
p4: (pattern (dots b c))
Pattern Unification fails.
```

Figure 3-27: Pattern unification

3.3.10. Unification

It is possible to define a natural partial order over the set of FDs. An FD *X* is more specific than the FD *Y* if *X* contains at least all the features of *Y* (that is $X \subseteq Y$). Two FDs are compatible if they are not contradictory on the value of an attribute. Let *X* and *Y* be two compatible FDs. The unification of *X* and *Y* is by definition the most general FD that is more specific than both *X* and *Y*.

```
   ((subject ((lex "John")))
    (verb ((lex "run"))))
```

unified with

```
   ((subject ((number singular)))
    (adverb ((lex "quickly"))))]
```

yields:

```
   ((subject ((lex "John") (number singular)))
    (verb ((lex "run")))
    (adverb ((lex "quickly"))))].
```

Figure 3-28: Example of unification

An example of unification is given in Fig.3-28. As mentioned in Sect.3.2.2 (p.56, when properties are simple (all the values are atomic), unification is very similar to the union of two sets: $X \cup Y$ is the smallest set containing both *X* and *Y*. There are two problems that make unification different from set union: first, in general, the union of two FDs is not a consistent FD (it can contain two different values for the same label); second, values of features can be complex FDs. The mechanism of unification is therefore a little more complex than suggested, but the functional unification mechanism is abstractly best understood as a union operation over FDs.

Figures 3-29 and 3-30 show a simple version of the unification algorithm. The main function of the algorithm is **unify(input, grammar)**. The figures also give a detailed description of important auxiliary functions. **Unify** calls first **graph-unify** on its arguments, performing a first sweep of unification through the input. Then, it identifies the constituents of the result of the first sweep (the Cset), and recursively unifies each of them. All the constituents of the total-FD, at all levels are therefore traversed in a breadth-first manner[21] and unified with the grammar, each according to its grammatical category. It is possible to describe recursive structures in the FU formalism. Note that **Graph-unify** is non-deterministic.

[21]The algorithm listed shows a depth first traversal of the constituent structure for ease of presentation.

```
Unify(input, grammar, path)
        input   : an FD with no disjunctions and no meta-fds.
        grammar : an FD.
        path    : the location of input in the total-FD (initially {})
Let     total-FD =  graph-unify(input, grammar, {})
        cset     = the constituent set of total-FD
    for each constituent C appearing at path P in cset do
        let UC   = unify(C, grammar, P)
            replace-value(total-FD, P, UC, C)


Atom-unify(fd1, fd2, path):
        fd1 : arbitrary sub-fd of input
        fd2 : atom - element of A or NIL or a meta-fd
        path: level of fd1 in the total-FD
if (fd1 = fd2)   return fd1
   (fd2 = NIL)   return fd1
   (fd2 = Any)   if fd1 is a ground-value
                        return fd1
                 else mark Any-P[path] and return Any
   (fd2 = None)  if fd1 is a ground-value
                        fail
                 else return None
else fail
```

Figure 3-29: Pseudo-code for unification algorithm (part 1)

Graph-unify is the most complex function. It unifies two FDs at the top-level. Basically, **graph-unify** enriches the input FD with all attribute-value pairs of the grammar that are not already in the input and recursively unifies the values of pairs that exist in both the input and the grammar. **Graph-unify** traverses the whole grammar depth-first, and calls the appropriate specialized function when it reaches the leaves: **Atom-unify** is called when one of the FDs is an atom. It succeeds if the two FDs are equal; it also implements the semantics of the meta-fds; **Pattern-unify** is a specialized function to handle patterns; **Cset-unify** handle Csets. Most of the complexity of **Graph-unify** is due to the handling of conflations.

The LISP implementation of the algorithm must handle the backtracking and manipulation of the total FD during the process. Backtracking is mainly restricted to the function `alt-unify` which handles non-deterministic choices when unifying disjunctions. The data structure representing the FD must support the operations:

- replace(FD, path, old-value, new-value)
- enrich(FD, path, new-value)
- find-value(FD, path)

One of the difficulties in the implementation of unification is the handling of copying. Analysis of existing systems has shown that copying data structures can take as much as half of the time a unification system spends. To reduce copying, the FUF implementation uses the technique of *undoable modifications*: all enrichments and replacements are made in place in the total FD, without copying, through physical modification. But all modifications are also recorded in a special stack. Upon backtracking, all modifications are undone, and the previous state of the total FD is restored before trying a new branch.

```
Graph-unify(fd1, fd2, path)
        fd1  : sub-fd of input
        fd2  : an fd with possible disjunctions and meta-fds
        path : level of fd1 in total-FD
If atom(fd2) return atom-unify(fd1, fd2, path)
else for each pair of fd2, label:value2 DO
    if (label = ALT) alt-unify(total-FD, value2, path)
    else if there is no pair in fd1 of the form (label:value1)
        enrich(total-FD, label:value2, path)
    else there is a pair label:value1 in fd1
        if (label = PATTERN)
            let value be Pattern-unify(value1, value2)
                enrich(total-FD, label:value, path)
        (label = CSET)
         let value be Cset-unify(value1, value2)
             enrich(total-FD, Cset:value, path)
        (value1 is a path and value2 is a path)
         let p1 = the fd pointed to by value1 in total-FD
             p2 = the fd pointed to by value2 in total-FD
             value = Graph-unify(p1, p2, value1)
             enrich(total-FD, value1=value)
             enrich(total-FD, path=value1)
             enrich(total-FD, value2=value1)
        (value1 is a path and value2 is not a path)
         let p1 = the FD pointed to by value1 in total-FD
             value = Graph-unify(p1, value2, value1)
             enrich(total-FD, value1=value)
             enrich(total-FD, path=value1)
        (value1 is not a path and value2 is a path)
         let p2 = the fd pointed to by value2 in total-FD
             value = Graph-unify(p2 , value1, value2)
             enrich(total-FD, value2=value)
             enrich(total-FD, path=value2)
        (normal case: value1 and value2 are valid fds)
         let value = Graph-unify(value1, value2, path|label)
             enrich(total-FD, label:value, path)
```

Figure 3-30: Pseudo-code for unification algorithm (part 2)

3.4. Using FUF for General-purpose Applications

While FUGs are particularly well adapted to the definition of linguistic applications, the computational mechanism is general enough to serve to implement a large class of procedures. In fact, FUF has been used to perform many non-linguistic tasks. For example, in the COMET multi-media system, FUF is used to perform the task of media-coordination [McKeown et al 90], which consists in annotating a semantic representation to identify which parts are best expressed as text and which parts are best expressed as graphics. In ADVISOR II, FUF is used for a large part of the knowledge representation task.

I take in this section the view of FUF as a general purpose programming language. Since FUF is a declarative language, this perspective is useful to (1) characterize the expressiveness of the formalism and (2) provide a procedural interpretation of the FUF constructs. To illustrate the generality of the formalism, I show in this section how FUF can be used to implement list processing operations like `append` and `member`. A comparison between the FUF and the PROLOG implementations of these operations highlights the specificities of FUF and provides a procedural interpretation of most of the FUF constructs. This comparison explains why FUF is not an appropriate programming language for most traditional programming tasks, but is well adapted to the task of text generation.

Specifically, this section proceeds as follows: I first show the code for the append and member functions in both FUF and PROLOG. The encoding of lists in FDs is then discussed. Lists are non-trivial to represent in FDs because FDs do not rely on order and position, whereas lists are a data-structure entirely relying on order and position. The FD representation of lists is a good example of a recursive FD structure, and illustrates the power of the `cset` construct.

I then compare the basic elements of a programming language with the corresponding FUF constructs: variables, environment and procedures with argument passing. Finally, the main characteristic of the FUF processing model is discussed: the total FD, which is the result of the unification, explicitly represents the trace of the whole computation. The relevance of this fact for generation is discussed.

While the exercise of implementing list-processing functions in FUF is purely formal, and is mainly motivated by exposition purposes, the techniques illustrated here are actually used in the SURGE grammar (to deal with conjunction), and in the ADVISOR II system (in the grammar representing the topoi-base, to implement a form of rule chaining).

3.4.1. The Member/Append Example

The grammar shown in Fig.3-31 implements the `append` and `member` operations in FUF. This grammar is actually equivalent to the PROLOG program shown in Fig.3-32.

```
'((alt
   (((cat append)
     (alt append
     ;; First branch: append([],Y,Y).
     (((x none)
       (z {^ y})
       ;; This is to normalize the result of a (cat append):
       ;; it must contain the CAR and CDR of the result.
       (car {^ z car})
       (cdr {^ z cdr}))

      ;; Second branch: append([X/Xs],Y,[X/Z]):-append(Xs,Y,Z).
      ((alt (((x ((car any))))   ; this alt allows for partially
             ((x ((cdr any)))))) ; defined lists X in input.
       ;; recursive call to append
       ;; with new arguments x, y and z.
       (cset (z))
       (z ((car {^ ^ x car})
           (cdr ((cat append)
                 (x {^ ^ ^ x cdr})
                 (y {^ ^ ^ y}))))))
       (car {^ z car})
       (cdr {^ z cdr}))))

    ((cat member)
     (alt member
      (((x {^ y car}))
       ((y ((cdr any)))
        (cset (m))
        (m ((cat member)
            (x {^ ^ x})
            (y {^ ^ y cdr}))))))))))))
```

Figure 3-31: Append and Member in FUF

The PROLOG form is much more concise and readable, but the point of this comparison is to precisely characterize the computational model implemented by FUF, not to reimplement `append`. Inputs that invoke the procedures are formed as shown in Fig.3-33.

I now explain how this grammar works, highlighting the specificities of the FUF computation model. The following aspects are reviewed in turn, to characterize FUF as a general-purpose programming language:

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y)

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Figure 3-32: Append and Member in Prolog

In PROLOG:
```
member(1, [a,1,2]).
append([a | R], [H, d], Z).
```

In FUF:
```
((cat member)
 (X 1)
 (Y ~(a 1 2)))

((cat append)
 (X ((car a)))
 (Y ((car nil)
     (cdr ((car d) (cdr none)))))
 (Z nil))
```

Figure 3-33: Inputs to the list processing programs

- **Representation of complex data-structures**: lists are not a primitive FD data structure. Lists are, however, a very general data-structure (which can be used to encode many other standard data-structures such as trees and graphs). I first show how to encode lists as FDs, illustrating how such a complex recursive data-structure can be encoded as an FD.
- **Representation of variables**: there is no explicit notion of variable in FUF. Instead, every data object is characterized by a location within the total FD. I discuss how paths can be used to encode variables.
- **Environment**: programming languages maintain an implicit environment to establish the mapping variable name/value. In FUF, the environment is explicit. I discuss how the total FD can be interpreted as a classical environment.
- **Function evaluation and argument passing**: programming languages encode abstractions as functions with arguments and function application using argument passing. I discuss how the cset mechanism of FUF can be interpreted as a form of function definition and application.

These aspects cover all the essential attributes of a complete programming language.

3.4.2. Representing Lists as FDs

The first problem to handle lists with FUF, is to represent lists as FDs, since FUGs can handle only FDs, and lists are not a FUF primitive data type. Quite simply, lists are represented as an FD with two features, CAR and CDR (with names ala LISP). The representation is illustrated in Fig.3-34.

The car/cdr notation for lists is quite awkward to use. FUF includes some syntactic sugar to translate between the regular LISP notation and the FD notation, similar to the [] notation in PROLOG. The syntax uses the macro-character "~" to indicate list values, as shown in Fig.3-35.

The "~" notation can only be used for completely specified lists. If some elements are uninstantiated, then the list

```
The list (a b c) is represented by the FD:

((car a)
 (cdr ((car b)
       (cdr ((car c)
             (cdr none))))))

The list (a (b c)) is represented by the FD:

((car a)
 (cdr ((car ((car b)
             (cdr ((car c)
                   (cdr none)))))
       (cdr none))))
```

Figure 3-34: FD representation of lists

```
((cat member)        ((cat member)
 (x a)          <==>  (x a)
 (y ~(c b a)))        (y ((car c)
                          (cdr ((car b)
                                (cdr ((car a)
                                      (cdr none))))))))
```

Figure 3-35: Tilde notation for lists

must be encoded with the car/cdr notation. In addition, the `~n` notation can be used to refer to elements of lists represented as FDs. The path `{l ~4}` refers to the fourth element of the list `l`. It is equivalent to the path `{l cdr cdr cdr car}`.

3.4.3. NIL and Logic Variables

In the previous example FD, the list terminator signal, equivalent to the LISP atom `NIL`, is `NONE`. `NIL` in an FD means *anything can come here* whereas `NONE` means `nothing can come here`. `NIL` therefore plays a role similar to uninstantiated variables in PROLOG. This comparison is illustrated in Fig.3-36.

The PROLOG expression `[a X c]` can be represented by the FD:

```
((car a)                              ((car a)
 (cdr ((car nil)                       (cdr ((cdr ((car c)
       (cdr ((car c)          <==>              (cdr none))))))
             (cdr none))))))
```

The PROLOG expression `[a b | Xs]` can be represented by the FD:

```
((car a)
 (cdr ((car b))))
```

Figure 3-36: NIL vs. PROLOG variables

3.4.4. Environment and Variable Names vs. FD and Path

The notions of environment and variable in PROLOG or LISP correspond to the notion of *total FD* and path in Functional Unification. The total FD is the highest level FD, identified by the empty path {}. The total FD corresponds to the input to the unifier. This FD contains all the environment of a computation.

Variables are then just places or locations within the total FD. This is illustrated in Fig.3-37.

```
If the total FD is the FD corresponding to [a X c]

((car a)
 (cdr ((cdr ((car c)
             (cdr none))))))

The variable X can be referred to by using the path {cdr car}
```

Figure 3-37: Total FD as environment

The notion of the total-FD serving as a structured environment is particularly important and is at the basis of the FUF interpretation as a blackboard-like architecture presented in Chap.4.

3.4.5. Procedures vs. Categories, Arguments vs. Constituents

A program in FUF can be viewed as a collection of procedures, each procedure being represented by a category. In the member example, an input containing the feature `(cat member)` will be sent to the `member` procedure.

Procedures expect arguments and return results. There is no notion of input and output in unification. Instead, and as in PROLOG, a procedure is viewed as a relation constraining arguments and result. For example, the `member` procedure has two arguments, called X and Y and represented in FUG notation by the constituents X and Y of the `(cat member)`. Similarly, the `append` procedure has three arguments, X, Y and Z. Z can be seen as the ‘‘result’’ of the procedure, or in functional notation: Z = append(X,Y).

As in the corresponding PROLOG program, the FUF implementation of `member` and `append` is non-directional. All of the arguments can be partially specified, and the unification enforces the relation existing between them in any order: `append([a b], Y, [a b c])` binds `Y` to `[c]`.

In order to ‘‘call a procedure’’ in FUF, the grammar needs to identify a subconstituent. This is done by using the `cset` mechanism. To identify which procedure must be called, the constituent is unified with a `(cat <procedure-name>)` feature. For example, in the `member` grammar, the procedure `member` is recursively called by building the `m` constituent of `(cat member)` with the arguments `x` and `y` instantiated.

3.4.6. The Total FD Includes the Stack of all Computation

The main characteristic of the FUF process is that there is no notion of implicit environment besides the total FD. Therefore, when a program works recursively, all the local variables that are normally stacked in an implicit external environment are stacked within the total FD. At the end, the total FD contains the whole stack of the computation, and looks pretty complex. As an example the result of the simple call `append([a,b],[c,d],Z)` is shown in Fig.3-38.

From a computation perspective, the only elements of interest in this FD are the values of the constituents `CAR` and

```
((CAT APPEND)
 (X ((CAR A) (CDR ((CAR B) (CDR NONE)))))
 (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
 (Z
  ((CAR A)
   (CDR
     ((CAT APPEND)
      (X ((CAR B) (CDR NONE)))
      (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
      (Z
       ((CAR B)
        (CDR
          ((CAT APPEND)
           (X NONE)
           (Y ((CAR C) (CDR ((CAR D) (CDR NONE))))
           (Z ((CAR C) (CDR ((CAR D) (CDR NONE)))))
           (CAR C)
           (CDR ((CAR D) (CDR NONE)))))))
      (CAR B)
      (CDR
        ((CAT APPEND)
         (X NONE)
         (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
         (Z ((CAR C) (CDR ((CAR D) (CDR NONE)))))
         (CAR C)
         (CDR ((CAR D) (CDR NONE)))))))))
 (CAR A)
 (CDR
   ((CAT APPEND)
    (X ((CAR B) (CDR NONE)))
    (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
    (Z
     ((CAR B)
      (CDR
        ((CAT APPEND)
         (X NONE)
         (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
         (Z((CAR C) (CDR ((CAR D) (CDR NONE)))))
         (CAR C)
         (CDR ((CAR D) (CDR NONE)))))))
    (CAR B)
    (CDR
      ((CAT APPEND)
       (X NONE)
       (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
       (Z ((CAR C) (CDR ((CAR D) (CDR NONE)))))
       (CAR C)
       (CDR ((CAR D) (CDR NONE))))))))
```

Figure 3-38: Stack of the computation in the total FD

CDR of Z, that is the value of the paths {Z CAR} and {Z CDR}. Explicitly encoding the whole history of the computation may therefore look wasteful for general purposes.

In contrast, this is a strong point of the approach for generation purposes. Indeed, in generation, one is not only interested in the resulting linguistic structure, derived from a semantic representation. Instead, the complete mapping from semantics to surface is of interest and is used throughout the generation process. FUF is therefore not only a rewriting system, but instead, it builds derivation structures explaining the rewriting from one level to another.

3.4.7. Summary

This section has described FUF as a general-purpose programming language, comparing it specifically with PROLOG. The following essential aspects of a programming language have been mapped to FUF constructs:

- The environment is explicitly encoded as the total FD. Variables are named by paths within the total FD. Uninstantiated variables correspond to the NIL value in the total FD.
- Recursive data-structures can be encoded using the cset construct.
- Functions are encoded as categories in the grammar. They are evaluated by placing the path to an FD of the appropriate category in the cset of an FD.
- Arguments to a function are encoded as features of the category encoding the function. Each argument is identified by the name of the corresponding feature (not by position).
- Functions are bi-directional and can be interpreted as relations between their arguments, as in pure PROLOG.
- The history of the computation is explicitly recorded in the total FD.

This analysis has provided a procedural interpretation for the FUF computation model. It has also characterized which types of applications are adapted to the FUF model (basically, those where the history of the computation is as important as the result of the computation and where a bidirectional computation may be required). It has also demonstrated techniques to write recursive grammars which have been reused in SURGE and in the content determination component of ADVISOR II.

3.5. Features and Limitations of the Original FUG Formalism

As I have developed larger and larger grammars using the original FUG formalism, I have come to appreciate its main features and encountered some of the limitations of the formalism. The features I have found most useful are:

- Uniform representation and simple mechanism to manipulate it. This allows the manipulation of linguistic, semantic and pragmatic information at all levels without constraints.
- Bidirectionality: no distinction between decision and realization.
- Use of partial information: there is no need to pre-declare fixed arities while prototyping grammars, inputs can be partially specified and can contain a mix of semantic and linguistic constraints.
- Flexible order of decision making: the grammar specifies a default order of decisions, but if there is some unforeseen dependency between decisions for a certain input configuration, the unifier can handle it through search and backtracking. As a consequence, the grammar writer is freed from the difficult task of ordering decisions for all situations.

In addition, the unification engine underlying the formalism is simple and small, leading to the development of a compact, responsive and easily manageable environment for grammar development.

Using this original formalism, I have developed a medium-size generation grammar inspired mainly from systemic linguistics as presented in [Halliday 85]. The grammar has been used as the realization component of the COMET multimedia project, and is documented in [McKeown et al 90] and [McKeown & Elhadad 91].[22] It has a structure very close to NIGEL's grammar. This realization grammar plays a role similar to Mumble86 in a generation system: it requires fully lexicalized input (*i.e.*, all open-class words are specified, and these lexemes are the primary means through which content is specified). The structure of the input is specified in terms of semantic functions but the heads are lexical, so the overall input structure is isomorphic to the linguistic structure in the sense that the semantic arguments of each lexical head are the same as the arguments of the corresponding syntactic head but possibly with

[22] Actually, the COMET implementation uses one of the extensions described in the next chapter: the `index` construct.

different labels. An example of input to this grammar is shown in Fig.3-39 and illustrates how ‘‘semantic’’ it is.[23]

```
((cat clause)
 (process ((type action)
           (transitive-class bitransitive)
           (dative-prep "to")
           (lex "give"))))
 (agent ((lex "John")
         (np-type proper)))
 (affected ((lex "book")
            (np-type common)
            (definite no)
            (describer ((lex "blue")))))
 (benef ((lex "Mary")
         (np-type proper)))
```

Figure 3-39: Input to the COMET grammar: *John gives a blue book to Mary*

While developing such large grammars, some of the limitations of the original FUG formalism became apparent in the context of practical systems. In particular, the following issues became important as the grammar grew:

- **Efficiency problems**: as the number of decisions made by the grammar grows, and the input is made more semantic and less syntactic, the processing time can become high. In most of the cases, the generation of a fully lexicalized input can be done in less than 2 seconds on a Sparcstation. But, the real problems occurred when trying to include lexical choice within the grammar. At this stage, order of decision making becomes more of an issue, and efficiency issues must be addressed.
- **Expressiveness and readability:** certain constraints became very difficult to express using only pure functional descriptions and standard unification only. In addition, for a whole class of grammatical constraints, the standard FUG formalism imposed inefficient and unreadable solutions.
- **Modularity and interaction with external knowledge sources:** to perform lexical choice, it became clear that external knowledge sources have to be queried at unification time, and that it is not practical to decide in advance what knowledge needs to be included in the input to the realization component. The original FUG mechanism does not support such interleaving of content planning and realization.

So while the original FUG formalism served us well to develop a realization component with roughly the same scope as MUMBLE and NIGEL, it also appeared to be too limited to address a more extended coverage. In particular, the decision to include lexical choice within the scope of the realization component increased the complexity of the task in such a way that new tools were required. I present in the next chapter a series of extensions that I have designed and integrated in the FUF system that allow it to address these more demanding tasks.

This chapter has mainly consisted of background material, but it collects information which is not available at this level of detail elsewhere. In addition, it has also presented two main contributions:

- The description of a complete implementation of the FUG abstract formalism, with the definition of the following points in more details than was previously available: a clarification of the definition of relative paths, the possibility to mix the structural notation (embedded lists) with the equation notation (with paths on the left-hand side of features), and a specification of the top-level flow of control of the unifier through the constituent structure.
- A procedural interpretation of the formalism, and a complete analysis of FUGs as a general-purpose programming language.

[23] When comparing this input with the one shown in Fig.3-1, which is the input for the same sentence for the SURGE grammar, note that this input has three constituents corresponding to the three syntactic constituents, subject, object, iobject. In contrast, the input in Fig.3-1 has four semantic constituents which are mapped onto three syntactic constituents in a non-isomorphic way.

Chapter 4

Extending FUF: Building a Practical and more Complete Generation Package

This chapter describes the extensions that I have added to the original FUG formalism to turn it into a practical generation environment. The resulting system is called FUF (for Functional Unification Formalism). The extensions have been designed to address problems arising during the development of a large portable generation grammar and specifically when implementing a lexical chooser in FUF. The task of lexical choice exhibits new complexities that are beyond the capabilities of formalisms designed to perform syntactic realization alone. I identify in this chapter aspects of the lexical choice task which make it difficult for the original FUG formalism. In particular, I identify a class of constraints called *floating constraints* which can interact in a complex manner with the rest of the generation process. Floating constraints appear in lexical choice because the lexical chooser must map two non-isomorphic structures: a conceptual structure with a linguistic structure. Thus, for example, several conceptual elements can be realized by the same lexical item (merging), and the same conceptual element can be realized at different syntactic levels (cross-ranking). Handling floating constraints efficiently is critical to perform lexical choice. I show in this chapter that the original FUG formalism was not well adapted to perform this task.

More generally, the problems met when extending FUGs to handle lexical choice and support a large syntactic realization grammar fall into three categories:

- **Efficiency issues**: when lexical choice was integrated into a grammar written in FUF, and floating constraints were considered, new challenging types of search configurations had to be solved by the unifier. The original chronological backtracking mechanism proved too inefficient to handle such tasks. I have therefore developed four new control tools that dramatically improve FUF's efficiency.
- **Usability issues**: I have found that the expression of common linguistic constraints, such as taxonomic relations between features or the expression of completeness constraints on a constituent, are difficult or impossible to express correctly in the original FUG formalism. Furthermore, as the grammar grows, especially when lexical information is included in the grammar, its organization, readability and maintainability become important. I have addressed these problems by developing a comprehensive type mechanism for FUGs. Types address the limitations of the original formalism in an elegant manner. They also allow for a more concise organization of the grammar, taking advantage of inheritance, improve readability and ease grammar maintenance.
- **Modularity issues**: as more of the generation process is encompassed within the scope of the FUF formalism, certain constraints have become difficult to express using only the original FUG formalism. Thus, interaction with external knowledge sources such as the knowledge base or the user model becomes necessary when dealing with lexical choice. I have developed a tool that allows the interleaving of decision making in FUF and in external knowledge sources while leaving the overall control to the FUF grammar. This facility allows the development of a generator in a modular architecture where the total FD being constructed plays a role similar to a blackboard in a blackboard architecture. Another consequence of the growing size of the grammar is the difficulty of maintaining it in a consistent state. I have developed a syntax for writing grammars in a modular way, allowing the user to define abstract alternations and conjunctions and reuse them by name in different contexts.

I present three classes of extensions, addressing each one of these issues in turn. Section 4.1 describes some generation problems which necessitate complex searching, identifying the class of floating constraints, and introduces the four new control tools implemented in FUF: indexing, goal delaying, dependency-directed backtracking, and conditional evaluation. Section 4.2 presents types in FUF, motivating the use of types with examples from the grammar and introducing the three varieties of types implemented in FUF: value typing,

constituent typing and procedural typing. The last section shows how FUF has been made modular, both by the development of a modular syntax and by allowing interaction with external processes that are not unification-based.

4.1. Control and Efficiency Issues

Generation is the process of making decisions to satisfy some communicative needs. Determining the order in which these decisions are made defines the problem of control. In this section, generation is considered as a search problem, where a configuration of linguistic devices must be found to satisfy an input configuration of constraints. Control strategies for text generation have received little attention in previous research. One of the reasons is that generation formalisms have most ofen been applied to relatively simple search problems: syntactic realization of completely lexicalized input. The main motivation for this section is that when lexical choice becomes part of the realization process, simple implicit search strategies are not sufficient anymore. In this section, I explain why the consideration of lexical choice in the realization process makes the search problem much harder by identifying a class of input configurations called floating constraints. Floating constraints appear when a conceptual input structure must be mapped to a non-isomorphic linguistic structure. I explain in this section what is the source of this non-isomorphism. The processing of floating constraints can trigger expensive backtracking when the simple search strategies used for syntactic realization alone are used.

This section presents tools I have defined and implemented in FUF to address the problem of control and make generation more efficient. The goal is to add knowledge to the grammar so that the interpreter can dynamically order decisions in an efficient way, and thus avoid an expensive blind search, even in the presence of floating constraints, when lexical choice is performed.

One trivial way to provide this knowledge is to explicitly order decisions in the grammar. This is the only approach available in NIGEL and MUMBLE for example. This is also the default strategy in FUF: the order in which decisions are written in the grammar is the order in which they are evaluated.

There is, however, a problem with this approach: there is no single order of decision making which can fit all the input configurations. In particular, I have characterized the class of floating constraints, which demand a flexible order of decision making. Floating constraints and their impact on control are discussed below. Other researchers have identified input configurations which require a flexible control. For example, Danlos has investigated some problems with pronominalization in French [Danlos and Namer 88]. One of the criteria used to decide whether to use a pronoun as opposed to a full referring expression is that the pronoun does not create ambiguity. In general, pronouns carry gender and number information through morphological inflection. So if gender and number can uniquely identify a referent in a discourse situation, then a pronoun can be considered. The decision to use a pronoun, according to all ‘‘delicacy’’ ordering proposed in the literature, should occur very early in the realization process. In French, however, object pronouns are placed before the verb, and when the verb starts with a vowel, there is an obligatory effect of elision, as in *Jean aime Marie* vs. *Jean l’aime* (where *l’* is the elision of *la*). The problem is that an elided pronoun loses the mark of gender (both *le* and *la* become *l’*). In this case, this morphological mechanism creates an ambiguity that makes the early pronominalization decision inappropriate. One solution is to delay the pronominalization decision until the morphology component has gathered enough information. But the morphological decision also depends on the syntactic function of the pronoun (the phenomenon only affects direct objects). This is an example of input configuration where semantic, syntactic and morphological factors affect the same decision in a complex way. Handling this situation in an efficient way would lead a grammar designer to force decisions in an order different from the one usually prescribed for the general case. So if decisions cannot be ordered dynamically by the grammar formalism, this would require duplication in the grammar, forcing the same decisions to be described twice in different orders.

Another issue is that it is desirable to free the grammar writer from the task of optimizing a grammar for a particular computational model. This is related to the criterion that grammars should be as declarative as possible. As a consequence, the FUF approach is to allow the grammar interpreter to search for a solution by committing non-deterministically to decisions and allowing for backtracking. Search, however, must be minimized. So the task of intelligent control is to dynamically plan an efficient order of decision making for a given input configuration.

The approach followed in this work has been to add annotations to the grammars that can be used by the unifier to improve performance. In a sense, this annotation approach is similar to the ‘‘optional type declarations’’ in

COMMON LISP, since they are optional, and only serve to optimize performance. An important constraint on the design of these annotations is that they do not change the semantics of the grammar, but uniquely the order in which the unifier processes it. Since one of the most important characteristics of the FUG formalism is that it is monotonic, decisions can be made in any order and the eventual result remains the same. All the control annotations are applied to disjunctions and not to conjunctions. From the measurements made on FUF working on large grammars, it was found that optimization of conjunctions was not necessary, as the average length of conjunctions over the course of a unification is quite small (on the order of 5) and thus time spent processing them is small. In contrast, the overhead of dealing with disjunctions is quite high, since it involves the management of non-deterministic constructs. Such control annotations express knowledge about the grammar and must be added by the grammar writer.

In this section, I first present the default flow of control used in FUF, which is top-down breadth first, and compare it to the bottom-up approach advocated in some generation work [Van Noord 90] and to the semantic-head-driven algorithm proposed in [Shieber et al 90]. I then introduce each of the four control annotations that I have defined in FUF to customize the search process to a specific grammar, explaining how they solve real problems met while developing a large grammar and how they interact with each other.

4.1.1. Top Down Control in FUF

The standard control strategy in FUF is to traverse the constituent structure in a top-down, breadth-first manner and in each constituent, to traverse the disjunctions depth-first, with chronological backtracking. Within each level of the constituent structure, the constituents are traversed in the order in which they are declared by a `cset` feature in the grammar.

When unifying an input with a grammar, two traversal strategies characterize the flow of control: the order in which the disjunction branches are tried in the grammar and the order in which the constituent structure of the total FD is traversed. The constituent traversal can be thought of as an outer loop where the body of the loop consists of unifying one constituent with one branch of the grammar dealing with its category.

The justification for the depth-first traversal of the disjunctions is that it makes the order in which the branches of the disjunctions are written significant. This control flow and the ordering effect it implies are similar to the PROLOG evaluation strategy, which is easy to understand and predictable.

For the constituent traversal order, the justification is more complex. The first intuition is that in a top-down traversal, the most important decisions are made before the details. So, for example, determining the structure of a clause, with its complements and adjuncts is decided before each complement is generated, and thus a top-down strategy would seem to follow the order of ‘‘delicacy’’ in the grammar. But beyond this intuition, a more detailed analysis is required: first, it is not clear what structure is being traversed top-down, since the input contains semantic information, but the linguistic constituent structure is being built incrementally, and not necessarily top-down; second, there are well-known problems with a purely top-down strategy that need to be examined. Finally, since the FUF mechanism to specify the constituent structure (cset) can contain arbitrary paths, and, therefore, the daughters of a constituent are not necessarily embedded features, a more precise definition of the flow of control is needed.

To simplify the discussion in this section, I assume that the input contains only semantic information, forming a logical form, and no linguistic information is included (no mixed input). The problem of controlling constituent order traversal is to decide how the logical form is traversed and at what point linguistic constituents are added to the linguistic structure. Early approaches to the task were derived from parsing formalisms (ATNs and DCGs) and used the natural top-down control these formalisms encourage. The main problem with a top-down regime is that it does not terminate on left-recursive rules. For example, the following grammatical rule allows the generation of possessive NPs as the determiner group of an NP, in a notation close to DCGs:

```
NP/LF(Head, Possessor) -> NP[possessive yes]/Possessor, NOUN/Head.
```

Each node in this rule is of the form `Linguistic-structure/Logical-form`. The rule determines how the logical form is mapped to an appropriate linguistic structure. The problem with this rule is that, in a top-down regime, it can be invoked again and again when the `Possessor` variable is not instantiated. Since many reasonable linguistic analyses rely on such left-recursive rules, this non-termination of the top-down regime is a serious problem. The problem is that the grammar rule can be applied in a non-goal-oriented manner, as noted in [Van

Noord 90]: a linguistic structure can be built for a non-existent logical form. Three types of solutions have been proposed to address the problem, as reviewed in [Shieber et al 90] and [Van Noord 90].

A first approach is to maintain the overall top-down flow of control and address the non-termination problem by adding information to the grammar. Dymetman and Isabelle propose to use explicit annotations in the grammar, added by the grammar writer, to control the order in which rules can be used [Dymetman & Isabelle 88]. Since one of their goals is to use the same grammar for both parsing and generation, the ordering of the rules in the text of the grammar is not sufficient since the order is different when parsing and generating. [Wedekind 88] proposes to use a specialized form of goal delaying which blocks rules until the logical form argument is instantiated, as in [Naish 85]. No local reordering of the rules, however, can prevent non-termination in the case of the ''complement rule'' [Shieber et al 90, p.32]:

```
vp(Head, Subcat)/VP --> vp(Head, [Comp/LF|Subcat])/VP, Comp/LF.
```

This rule describes how the subcategorization frame of a verb gets filled by adding complements. (A similar type of rule is used in HPSG, GPSG and LFG.) The problem is that this rule will have to be invoked when the LF variable is not instantiated. Since the verb is not yet known when this rule is invoked, the `Subcat` variable is not instantiated and, therefore, the `LF` variable will always be non-instantiated in a pure top-down regime. Therefore, all ordering or delaying schemes will require the `VP` node to be expanded first. But if the `VP` node is expanded first, the same rule can be applied indefinitely. This example illustrates how a linguistically motivated rule can prevent any form of top-down regime from terminating.

Other researchers have proposed to abandon the top-down regime and adopt a bottom-up approach ([Shieber 88] and [Van Noord 90]). There are two known problems with a bottom-up approach: it can create artificial non-determinism and it imposes a strong constraint on the grammar rules, known as *semantic monotonicity* [Shieber 88]. An example of artificial non-determinism is given by handling of the case feature in pronouns. The case (accusative or nominative) or a pronoun depends on its function within the clause. If the pronoun is first generated before knowing where it will fit in the clause, the case is not determined. A bottom-up grammar does not allow grammatical information to flow from linguistic heads to their dependents in a natural way. The other problem is that to make the bottom-up regime goal-driven, the grammar writer must enforce the constraint that the logical form of every sub-phrase that will become part of the generated sentence must subsume some part of the input logical form. In this way, subphrases are only added when they ''consume'' a part of the input. The problem with this constraint is that it makes it difficult to account for words with empty semantic content like *it* or *there* (expletive expressions), for verb particles and for idioms.

Finally, the most recent approach is embodied in the semantic-head driven generation algorithm (cf. [Shieber et al 90] and [Van Noord 90]) which is a sort of composition of top-down and bottom-up. The underlying idea is to identify the semantic head in the right-hand side of certain rules, connect the head to a lexical entry, and chain rules bottom-up from the lexical realization of the semantic head, and, recursively, top-down from the other constituents. Rules in the grammar are marked as either chain rules (the ones used for bottom-up traversal) or non-chain rules.[24] A chain rule is one where the semantics of some right-hand side element is identical to the semantics of the left-hand side. The right-hand side element is called the semantic head of the rule. For example, the complement rule discussed above is a chain rule whose semantic head is the vp element of the right-hand side. The following rules are nonchain rules, because no element in the RHS has a semantics identical to the semantics of the LHS:

```
sentence/decl(S) --> s(finite)/S.
vp(finite, [np(_)/O, np(3-sing)/S])/love(S,O) --> [loves].
```

Chain rules are used in the bottom-up phase of the traversal. Nonchain rules are used in the top-down traversal, which cannot cause non-termination because the LHS of the rule cannot appear in the RHS by definition. The algorithm starts at a node root. It proceeds by applying any applicable nonchain rule top-down. A nonchain rule is applicable if the semantics of its LHS matches that of the root node. The nonchain rule identifies a new syntactic node in the emerging syntactic tree, called the pivot, which realizes the semantic head of the root. The problem with applying a nonchain rule immediately is that the pivot is not connected to the root syntactic tree. The rule has only be selected by considering the semantic form of its LHS with that of the root node, not the syntactic tree. A connected syntactic tree needs then to be constructed to link the pivot to the root node. This is achieved by chaining through the chain rules bottom-up from the pivot node to the root node, fleshing out the syntactic tree in the process.

[24] This marking is done automatically by a PROLOG program.

Each time a chain rule is applied, the algorithm recurses on the daughter nodes the rule introduces which are not its semantic head.

An example derivation is shown in Fig.4-1, where the arrows indicate in which direction the generation step was performed (up or down) and the numbers indicate in which order the nodes were generated. The example sentence shows how the particle *up* in *John calls friends up* is generated even though the rule for it does not cover any part of the input logical form (something impossible with a purely bottom-up generator) and how the ‘‘complement rule’’ is handled without looping.[25]

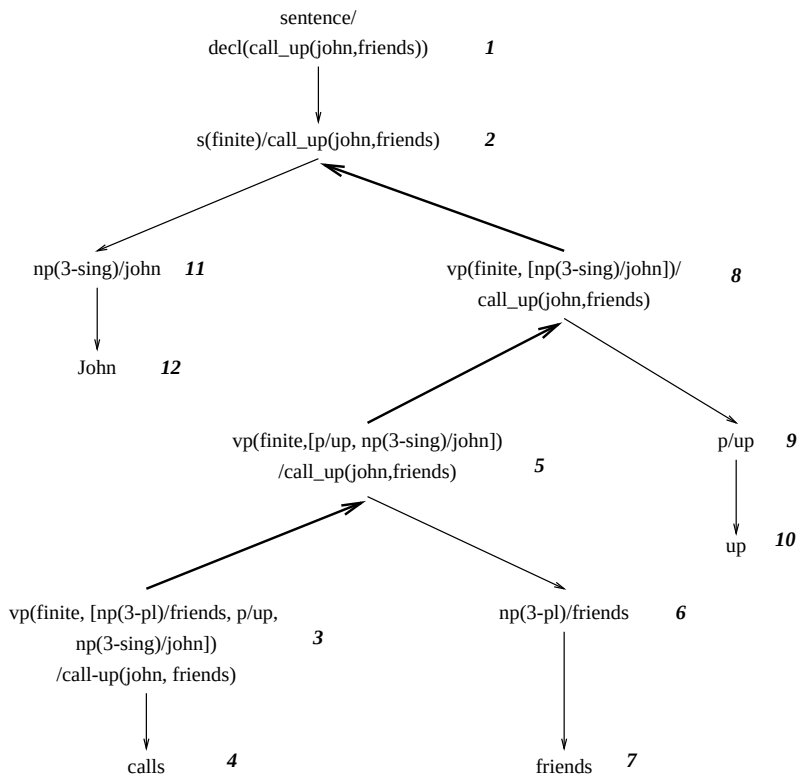


Figure 4-1: Analysis tree using semantic-head driven generation [Shieber et al 90]

In FUF, I use a variation of a top-down regime with freezing (which causes rule reordering). Because in the current system lexical choice and syntactic realization are separated in two successive processes, this control regime needs to be evaluated for both tasks separately. To evaluate how top-down with freezing behaves for syntactic realization, note that, in the above survey of control strategies, the only practical problem found with the amended top-down regime allowing rule ordering was with a rule similar to the complement rule. The problem is that this rule builds a syntactic structure without relying on lexical information: it can therefore generate a subcategorization frame which does not match any frame available in the lexicon. The basic observation is that eventually all the syntactic structure must be accounted for in terms of the subcategorization frames of lexical heads. Since the input to SURGE, the syntactic realization grammar, is fully lexicalized, the verb is already instantiated, and its subcategorization frame is also instantiated. In this case, the complement rule, or any similar rule building a syntactic structure, cannot loop (if the left-recursion problem is handled correctly as explained below). So for syntactic realization, there does not seem to be critical problems with the top-down control regime with freezing.

For lexical choice, the top-down regime with freezing does not cause problems under the following assumptions: the role of the lexical chooser is to map a conceptual structure onto a linguistic structure characterized by lexical heads

[25]In the figure, the predicate for VPs is vp(finite, Subcat) where Subcat is the subcategorization frame of the vp.

and their arguments. The goal of the lexical chooser is to cover an input conceptual form. The approach in FUF is: select a head in the conceptual form[26], lexicalize it (which includes mapping the semantic arguments to complements), then recurse top-down on each complement. The overall flow remains top-down, with the constraint that the semantic head must be lexicalized before the top-down traversal can proceed, because only the lexical head can determine where to go down.

When the lexicalization of the head depends on the conceptual type of its arguments, for example, the concept *ingest* can be lexicalized as *drink* if the object is liquid and *eat* if it is solid, the lexicalizer has access to these types when selecting a head verb, so the choice can be done before lexicalizing the arguments (subject and object). If the lexicalization of the head depends on the linguistic category of its arguments, then a delay must be introduced. Consider for example the three following quasi-paraphrases, realizing the logical form `demand(Karen, elect(_,he))`:

(1) Karen demands him to be elected.
(2) Karen demands that he be elected.
(3) Karen demand his election.

I consider here that these three occurences of *demand* actually correspond to three distinct lexical entries, with different subcategorization frames. In (1), *demand* has two objects - *him* and *to be elected*. In (2), it has one object, the *that*-clause, and in (3), it has also one object, the nominalization. The choice between these three entries is constrained in part by how the conceptual form `elect(X,Y)` is lexicalized: as a clause, an NP or an NP plus a clause.[27] In this case, the delay is used as follows: the choice between the three forms of *demand* is blocked until the syntactic category of its arguments is known. Recall, that the control regime would lead to possible looping if forced to lexicalize a constituent for which the conceptual form is not instantiated. In the current case, lexicalizing the ''object'' of *demand* without mapping it to a conceptual element would cause trouble. To avoid this, the lexicalizer first recurses on the conceptual element `elect(X,Y)`, and when the result of this lexicalization is determined, it is attached to the syntactic form by choosing the appropriate lexical entry for *demand*. This attachment stage is similar in spirit to the bottom-up chaining used in the semantic-head driven algorithm. Note also that in the FUF strategy, semantic heads are directly selected at each step, therefore using the terminology of the semantic-head driven algorithm, the lexical chooser only uses nonchain rules in a pure top-down regime. The simplification of the FUF control regime compared to the semantic-head driven algorithm is mainly due to the fact that lexical choice and syntactic realization are separated in two distinct stages.[28]

Two aspects of the basic FUF mechanism for specifying the constituent traversal order, in addition to the goal freezing mechanism described below, provide enough flexibility to implement this lexicalization strategy:

- the constituent structure can be declared incrementally and dynamically by adding elements to the cset of a constituent.
- daughter constituents can be located anywhere within the total FD since elements of the cset can be arbitrary paths. In particular, it is possible (although counter-intuitive) to declare that a constituent embedding the current constituent is one of its descendants in the constituent structure.

More details on how this control flow is implemented in FUF, using the control tools described later in this section, are provided in Chap.6.

Before proceeding to the more complex performance-enhancing control tools developed in FUF, I first introduce a tool which addresses the most obvious limitation of a top-down regime with left-recursive rules. The tool is the `given` meta-variable: it ensures that the top-down regime of FUF is goal-directed. `Given` is in a sense the dual of the `any` meta-variable of the original FUG formalism: while `any` requires a feature to be instantiated at the *end* of the

---

[26]This selection is based on pragmatic criteria similar to focus and information packaging principles. The selection process is detailed in Chap.6.

[27]Other pragmatic factors are obviously influencing the decision and the sentences have slightly different meanings.

[28]In general, the semantic-head driven algorithm is derived from parsing techniques that remain attached to the output string. Some of the ''problems'' mentioned in this approach have never been issues for formalisms developed purely for generation (as the three formalisms reviewed in Chap.3). For example, delaying morphological processing until after syntactic realization is presented as a ''new idea'' in [Shieber et al 90], when no other approach has ever been considered by pure generation formalisms. Similarly, performing lexical choice and syntactic realization simultaneously introduces most of the problems that the semantic-head driven algorithm addresses.

unification process, `given` requires it to be instantiated *before* unification starts. Thus, `given` is used to check that a constraint has been specified in the input. A `given` feature can only appear in a grammar, thus, `given` gives a different status to the two arguments of the unification function.[29]

```
((cat NP)
 (semr ((possessor GIVEN)))
 (cset (det head))
 (det ((cat NP)
       (possessive yes)
       (semr {^ ^ possessor}))))
 ...)
```

Figure 4-2: A left-recursive rule in FUF using given

To illustrate how `given` solves the problem of the left-recursive rules, consider how the possessive NP rule is implemented in FUF in Fig.4-2. This FD implements the rule shown above in DCG format. To avoid using this rule when there is no possessor in the input (which is the step which leads to non-termination), the grammar contains a feature `(possessor GIVEN)`, checking that the semantic representation of this constituent does indeed have a possessor specified in the input. If none is specified, then the rule will fail, and the sub-constituent possessive NP will not get created.

When dealing with typed features, it is possible to impose bounds on how instantiated a feature is by using an extension of `given` called `under`. A construct of the form `(att #(under val))` unifies with a feature `(att x)` if and only if `x` is subsumed by `val` in the type hierarchy; that is, if `x` is more specific than `val`.

I now enumerate the control tools introduced in FUF which allow the grammar writer to make generation more efficient and address the complexities of the lexical choice task.

4.1.2. Indexing

The simplest way of controlling search is also one of the most effective. There are many occurrences in the grammar of disjunctions where the choice between the different branches of the disjunction can be determined based on the value of a single "key" attribute. For example:

```
(alt (
       ((person first)
        ...)
       ((person second)
        ...)
       ((person third)
        ...)
     ))
```

In this case, the `person` attribute can be used as a key to ‘‘jump’’ to the right branch when it is available. To indicate such cases, FUF allows the use of the `index` annotation:

```
(alt (:index person) ...)
```

This annotation is interpreted by the unifier as follows: if when the disjunction (alt) is unified the current input has an instantiated value for the key attribute (`person`), then based on this value the unifier will only consider the correct branch and will not use any backtracking point evaluating this disjunction. If the value of the key attribute is not instantiated yet, then the disjunction is unified normally. Such annotation is easily added in all grammars and it does boost the performance dramatically, as it avoids all the overhead associated with putting and removing backtracking points.

---

[29] This means that when `given` is used, unification is no longer symmetric, but it remains order-independent and monotonic.

The key attribute does not have to appear at the toplevel of the branches; embedded attributes at any path are legal keys. Furthermore, the key does not need to uniquely discriminate among the branches: if a disjunction has five branches, with respective values `{a,a,b,c,d}` on the key attribute, and the current value of the key attribute is `a` then the disjunction will remain but will be filtered to only contain the first two branches `{a, a}`. Unification will proceed as usual on these two branches.

In the current grammar of FUF, 222 of the 580 disjunctions are indexed. On a test of 150 cases generating clauses and noun phrases, the following measurements indicate the speedup provided by index.

Number of cases	150
Average # backtracking points	188
Standard deviation	61
Average speedup	24.80%
Standard Deviation	5.5%
Median speedup	24%
Maximal speedup	43%

All measurements are made using number of backtracking points. A backtracking point corresponds to a choice in a disjunction. The total number of backtracking points is in a way the number of questions the grammar asks to generate a clause or a noun group. Running time is approximately linear in the number of backtracking points, up to a threshold size of the input, after which other factors besides number of backtracking points impact on running time (in particular, depth of embedding and size of the total FD). In practice, single sentences do not reach this threshold size. The cases measured are typical sentences and required from 100 to 300 backtracking points, with an average 188. All 150 tests were run on a grammar containing no index annotation first, then on the same grammar with the index annotations. The speedup is measured as the difference between the number of backtracking points without index and with index. The average speedup is 24.80%. The most interesting feature of this optimization is that all input configurations are equally well affected (small standard deviation of 5.5). Note also that indexing is very close to Jacob's hashing-scheme in PHRED [Jacobs 85] but that it works without changing the overall FUF formalism and mechanism.

In conclusion, indexing requires very little effort from the grammar writer and reduces the running time of the grammar almost uniformly on average by one quarter.

4.1.3. Floating Constraints[30]

While indexing works well on all cases and provides a uniform speedup, the main problem of the non-deterministic search mechanism used by FUF is that in some occasions the search space can explode in exponential time. On the difficult configurations that trigger such expansive search, indexing would only take out a small fraction of an exponentially large number of backtracking points. More powerful control mechanisms aimed at controlling the search process when ''things go wrong'' are therefore required.

The first step in addressing the problem is to identify a class of input configurations which actually trigger exponential backtracking. I have identified a class of such input constraints that I call *floating constraints*. Floating constraints impose a difficult search procedure on the grammar and require advanced control tools to be satisfied in bounded time. I first define floating constraints in this subsection and then move on to present two control tools addressing the problem they pose to the unifier: dependency-directed backtracking and goal delaying.

The task of the generator is to map from a semantic constituent structure to a syntactic one. This task is difficult because, in general, these structures are not isomorphic:

---

[30]This section describes work done in collaboration with Jacques Robin. It is derived from [Elhadad & Robin 92].

> A combination of semantic elements can be expressed by a single surface element, or a single semantic element by a combination of surface elements. [Talmy 85, p.57]

I present in this section examples developed with Jacques Robin in the basketball domain. Robin's dissertation work in on report generation, and models the task of generating basketball games reports. The examples in this section are derived from his work [Robin 92a]. In this domain, for example, the clause pattern *X edged Y* conveys two semantic elements: a semantic predicate - X won a game against Y - and a manner qualification - the game was close. This non-isomorphism between syntactic and semantic structures is a pervasive phenomenon, as illustrated by Talmy's extensive cross-linguistic analysis of constructions expressing motion and causation ([Talmy 76] and [Talmy 83]).

Moreover, the same semantic element can be realized by syntactic elements at different linguistic ranks (*e.g.*, group, clause, sentence) For example, the low rating of a team (an argumentative orientation constraint) can be conveyed by a variety of syntactic constituents:

- Adjective (at the noun-group rank): *The hapless Denver Nuggets beat the Boston Celtics 101-99.*
- Verb (at the verb-group rank): *The Denver Nuggets stunned the Boston Celtics 101-99.*
- Adverbial (at the clause rank): *The Denver Nuggets surprisingly beat the Boston Celtics 101-99.*

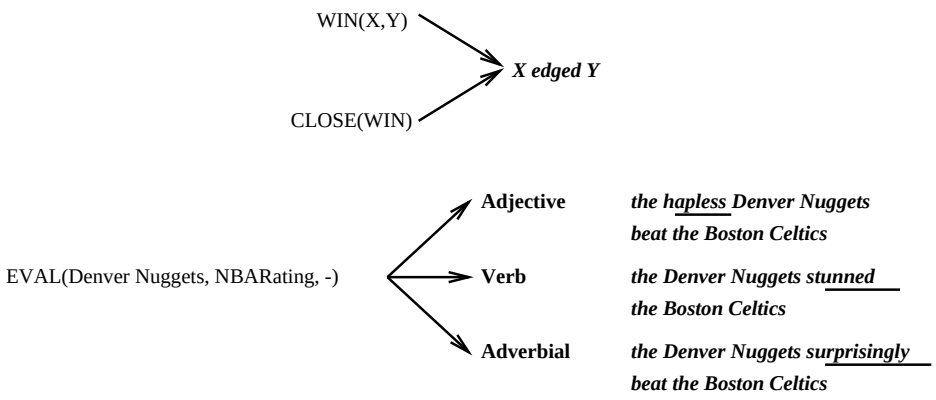


Figure 4-3: Non-isomorphism between semantic and linguistic structures

The situation is illustrated in Fig.4-3. I call such semantic elements *floating constraints*. I distinguish them from *structural constraints* such as semantic predications or references. Structural constraints require the presence of syntactic constituents at a given linguistic rank in the output and thus guide the mapping process from the semantic structure to the syntactic structure. For example, when an input event structure is mapped to a clause, the semantic predicate `c-win` determines how the semantic roles are mapped onto syntactic complements: `winner` to `subject` and `loser` to `object`. Figures 4-4 and 4-5 illustrate the difference between the two types of constraints.

The top-down regime implemented in FUF handles structural constraints efficiently because backtracking is circumscribed to the unification of the grammar with a *single* input constituent. In contrast, the processing of floating constraints can be very inefficient because it can trigger non-local backtracking, cutting across linguistic ranks and requiring the re-unification of the grammar with *several* input constituents. To illustrate this problem, consider a system reporting on the results of a basketball game and an input containing the following three constraints:

- Semantic Predication: convey that the Denver Nuggets defeated the Boston Celtics by a 101-99 score.
- Manner Qualification: convey that the game was tight.
- Argumentative Orientation: convey the low rating of the Denver Nuggets.

For example, the above input configuration of constraints is correctly satisfied by the following sentence: *The hapless Denver Nuggets edged the Boston Celtics 101-99.*

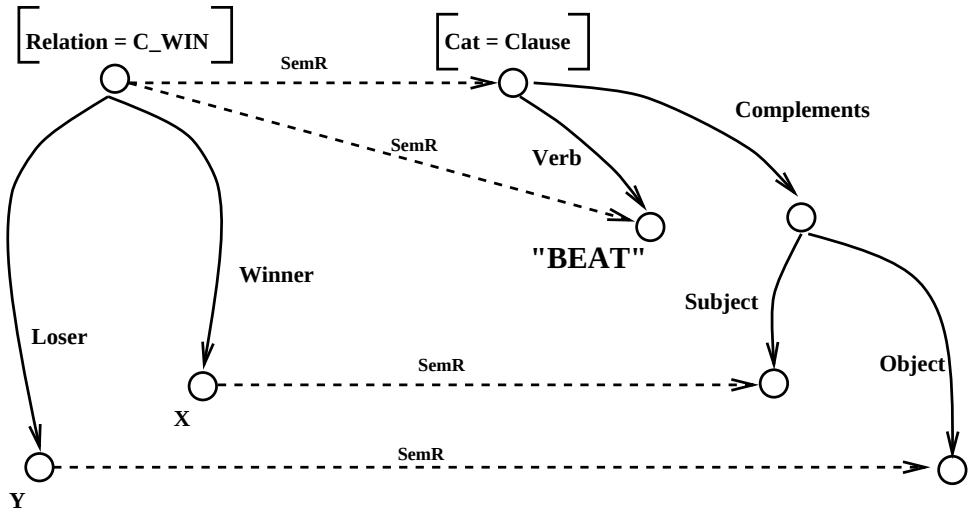


Figure 4-4: A structural constraint

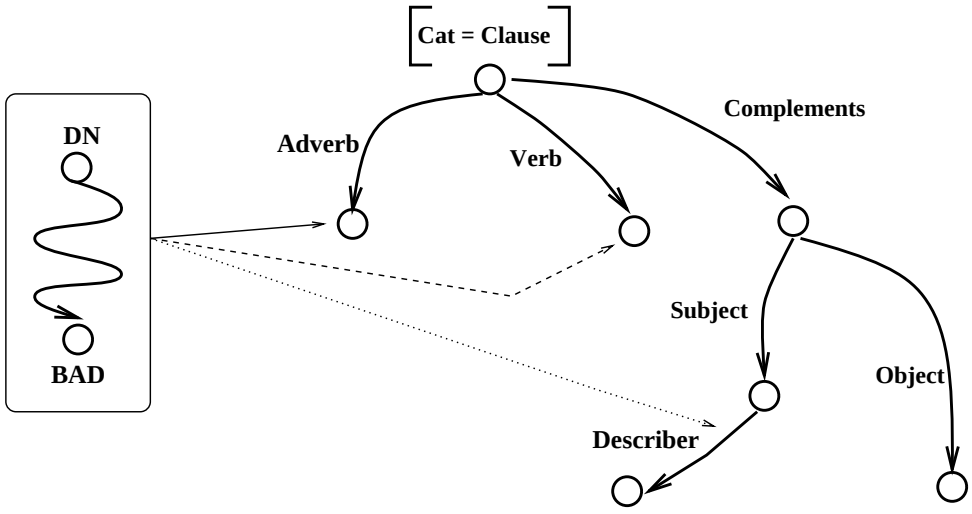


Figure 4-5: A floating constraint

But all these different linguistic devices cannot be freely combined, as illustrated by the following examples:

1. *? The Denver Nuggets narrowly stunned the Boston Celtics 101-99.*
2. *? The Denver Nuggets surprisingly nipped the Boston Celtics 101-99.*
3. *? Against all odds, the Denver Nuggets narrowly beat the Boston Celtics 101-99.*

In sentence (1), it is not clear which semantic aspect of the verb *stunned* is modified by *narrowly*: the expression of the game result or its unexpectedness. Similarly in (2), the modification of *nipped* by *surprisingly* is ambiguous. In

(3), the scope of *against all odds* is ambiguous: it could be either *narrowly* - in which case the Nuggets are presented as highly rated - or *beat* - in which case the Nuggets are presented as lowly rated.

The input FD shown in Fig.4-6 encodes the three constraints to satisfy. Figure 4-7 shows a fragment of a lexicon specifying the mapping between concepts and lexical items. The fragment shows how different verbs impose constraints on the features `AO` or `manner`, or no constraint for ''neutral'' verbs. The branch order in the `win-lex` alt enforces the stylistic preference for semantically rich verbs over neutral verbs with adverbials.[31] In addition, to avoid generating adverbials with ambiguous scope, the grammar enforces that (1) clauses contain a single adverb and (2) only neutral verbs are used in combination with adverbials.

```
(;; Semantic predication
 (sem-cat action) (concept c-win) (token t-win-666) (tense past)
 (winner ((sem-cat individual) (concept c-team) (name Nuggets)))
 (loser ((sem-cat individual) (concept c-team) (name Celtics)))
 (score ((sem-cat quantity) (concept c-game-score)
         (winner-score 101) (loser-score 99)))
 ;; Manner constraint
 (manner ((sem-cat quality) (concept c-tight)))
 ;; Argumentative constraint
 (AO ((sem-cat scale) (concept c-rating)
      (carrier {winner}) (orientation -))))
```

Figure 4-6: An input expressing two floating constraints

```
(...
 ((sem-cat action)
  (alt (index on concept)
    ;; Map the concept game-result to a verb
    (((concept win)
      (alt win-lex (:bk-class (AO manner))
        (;; The winner's rating is poor
         (({AO} ((concept c-rating) (carrier {winner})
                 (orientation -) (conveyed yes)))
          (lex ((alt ("stun" "surprise"))))
          ({adverb} none))

         ;; The victory is narrow
         (({manner} ((concept c-tight) (conveyed yes)))
          (lex ((alt ("edge" "nip"))))
          ({adverb} none))

         ;; Default neutral verbs
         ((lex ((alt ("beat" "defeat" "down"))))))))
       ... ))))
```

Figure 4-7: Choice of a verb in the lexico-grammar

Consider the realization of the semantic input in Fig.4-6 with the grammar in Fig.4-7. FUF's top-down regime allows it to map the structural constraints to syntactic constituents right away: first the semantic predicate to a verb-group, and then the roles `winner`, `loser` and `score`, to `subject`, `object` and `adjunct` respectively. In contrast, the mapping of the ''floating'' constraints `AO` and `manner` must be delayed.

[31]FUF tries the branches of an `alt` construct in order. When no order is preferable, the construct `ralt` (standing for Random Alternation) whose branches are tried at random is used instead.

Figure4-8 illustrates this delaying mechanism by showing how the `manner` input constraint is handled by the grammar. The feature `manner-conveyed` is used to record the syntactic category of the constituent realizing the manner constraint. It remains `nil` as long as the constraint is not conveyed. The first branch of the `manner-adverbial` alt checks whether the manner constraint has already been handled by some other constituent. This check is implemented by the feature `(manner-conveyed any)`. This first branch delays the decision to use an adverb with the `any` construct. This gives other devices a chance to express the manner constraint. However if no other linguistic device can be found that satisfies the manner constraint, the grammar resorts to using an adverbial adjunct, by choosing the second branch of the alt. The feature `(manner-conveyed any)`, therefore, prevents the generation of semantically incomplete sentences like *The Denver Nuggets stunned the Boston Celtics 101-99.* The argumentative constraint is similarly handled with a feature `ao-conveyed`.

```
( ...
 (verb ((alt ...))) ...
 (AO ((alt ...))) ...
 (manner ((alt manner-adverbial (:bk-class manner)
           (;; Can be realized by other means - delay
            ((manner-conveyed any))

            ;; Map manner to an adverbial adjunct
            ;; and mark that manner has been realized
            (({adverb} ((synt-cat adverb) (concept {^ ^ concept})))
             (manner-conveyed adverb))))))
  ... )
```

Figure 4-8: Handling floating constraints

Let us now consider how the manner and argumentation constraints interact. In a top-down regime, the verb-group is first processed and the concept `c-win` is lexicalized. FUF is now traversing the lexicon fragment in Fig.4-7 and first chooses the verb *stun* which satisfies both the semantic predication and the argumentative constraint. It then maps the semantic constituents to syntactic functions and proceeds to the argumentative constraint. This constraint is already satisfied by the verb, so no modifier needs to be introduced.

At this point FUF attempts to take into account the manner constraint. It first delays the use of an adverb with the `any` construct and completes the traversal of the constituents top-down. It eventually checks the `any` construct and finds that the manner constraint has not been satisfied. Backtracking is triggered. Consider at this point the state of the backtracking-point stack: the whole grammar has been traversed, all the subconstituents processed. Basically, all potential backtracking points are on the stack. Figure 4-9 illustrates the state of the stack at this point. If FUF blindly backtracks, search is maximized. Since it cannot be known *a priori* where in the syntactic structure the floating manner constraint will fit, the decision whether to use an adverb must be delayed until the end of the traversal. There is, therefore, no way to detect failure before this point.

In general, floating constraints form a class that pose problems in terms of control in generation. I now describe two control tools implemented in FUF that allow the grammar writer to deal with the complexity of floating constraints.

4.1.4. Dependency-directed Backtracking

The first tool introduced to avoid the cost of a blind backtracking is the `bk-class` construct. It implements a version of dependency-directed backtracking [de Kleer et al 79] specialized to the case of FUF. The `bk-class` construct relies on the fact that in FUF, a failure always occurs because there is a conflict between two values for a certain attribute at a certain location in the total FD. In the example illustrating this section, backtracking is necessary because an equation requires that the value of the feature `{manner manner-conveyed}` be instantiated, but the actual feature is not. The path `{manner manner-conveyed}` defines the *address of the failure*.[32]

[32]In an FD, each embedded feature can be viewed as an equation between the path leading to the feature in the total FD and the feature value.

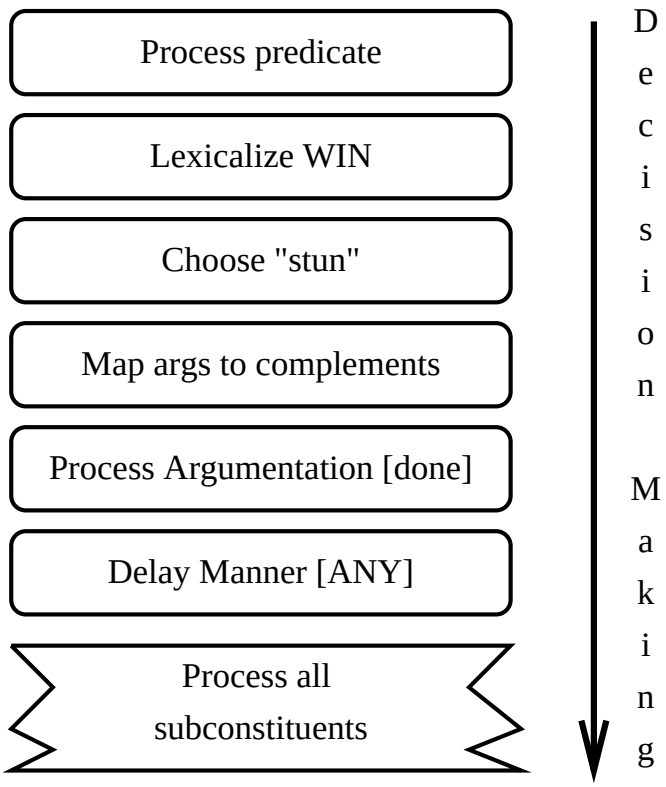


Figure 4-9: Stack of choice points after one pass through floating constraints

The idea is that the location of a failure can be used to identify the only decision points in the backtracking stack that could have caused it. This identification requires additional knowledge that must be declared in the FUG. More precisely, the FUG writer is first allowed to declare certain paths to be of a certain `bk-class`. Then, in the FUG, every choice points that correspond to this `bk-class` must be explicitly declared.

For example, the statement: `(define-bk-class manner {manner manner-conveyed})` specifies that the path `{manner manner-conveyed}` is of class manner. In addition, all `alts` that have an influence on the handling of the manner constraint are tagged in the FUG with a declaration `(:bk-class manner)` as shown in Fig.4-8.

When the unifier fails at a location of class `manner`, it *directly* backtracks to the last choice point of class `manner`, ignoring all intermediate decisions. In the example, when the `any` constraint fails, the unifier directly backtracks to the manner choice point in the grammar (Fig.4-8). If this last option fails again, it backtracks up to the choice of verb in the lexicon (Fig.4-7). Figure 4-10 illustrates the effect of using `bk-class`. The unifier therefore uses the knowledge that *only* the verb or the adverb can satisfy the manner constraint in a clause to drastically reduce the search space. But, this knowledge is *locally* expressed at each relevant choice point, retaining the possibility of independently expressing each constraint in the FUG.

In general, the determination of the address of failure is more complex and it is necessary to distinguish between *initial failures* and *derived failures*. An initial failure always occurs at a leaf of the total FD, when trying to unify two incompatible atoms. Failures, however, can also propagate up the structure of the total FD. For example, when unifying `((a ((b 1))))` with `((a ((b 2))))` the original address of failure is the path `{a b}`. When the unifier backtracks, it also triggers a failure at address `{a}`, which is not a leaf. This type of failure is called a derived failure. In the implementation of `bk-class`, FUF ignores derived failures and directly backtracks to the first choice point whose `bk-class` matches the last initial failure. Determining whether a failure is derived or initial can be difficult. In FUF, the following technique is used: I distinguish between *forward* and *backward* processing. Forward processing is the normal advance of the search procedure through the disjunctions in the grammar. Upon failure, backward processing starts, which means that the stack of the last decisions is unwound. Each time a new

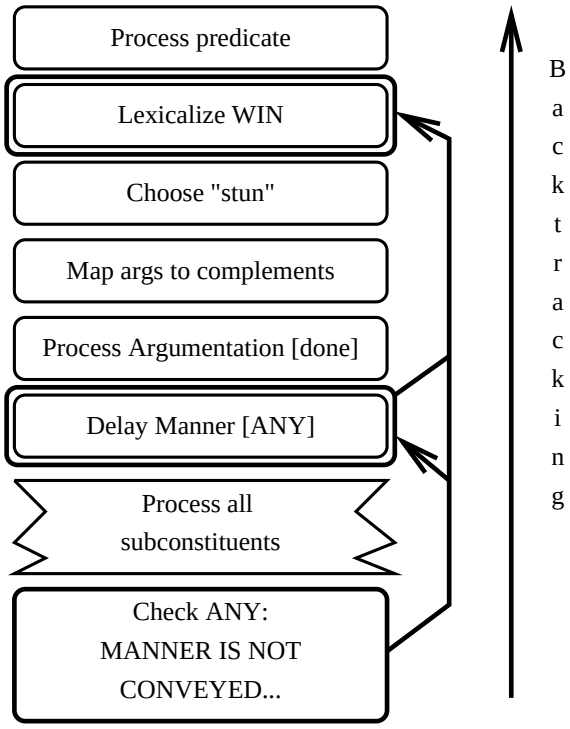


Figure 4-10: Effect of bk-class

disjunction is retried and restarted, forward processing could restart, but unification may fail again immediately, without making any changes to the total FD being processed. In general, backward mode continues until one of two events occurs: either a restarted alt succeeds and at least one change is made to the total FD (a feature is added to the input), or a failure occurs when unifying a grammar feature with a feature that was present in the original input, before unification started. The distinction between forward and backward processing and the way the address of failure is maintained is illustrated in Fig.4-11.

The figure shows the search space when traversing two successive alts in the grammar. Each arrow corresponds to either the choice of a branch in the alt or to a forced backtracking. Arrows 1 and 2 correspond to the first forward processing step and end up in a failure. This is an initial failure and the *address of failure* (which is by definition the location which caused the latest significant failure) is set to the current position in the total FD. The backward processing step starts and a second branch of the ALT is tried (arcs 3 and 4). This second branch fails immediately, before any change is made to the total FD. In such a case, the backward processing step continues, and the address of failure is **not** updated. The motivation is that the unifier is still backtracking for the same original reason: nothing changed in the grammar that makes the original failure less relevant. So the unifier keeps backtracking, and tries the third and last branch of the ALT (arcs 5 and 6). The same failure occurs again, and backward processing proceeds. The second ALT is exhausted and, therefore, fails, and the second branch of the first ALT is tried in turn (arcs 7, 8 and 9). At this time, two reasons can trigger a switch of failure address:

- Unification fails immediately, before any changes are made, but the failure is due to the incompatibility of a feature of the grammar with a feature that was originally present in the input before unification started. In such a case, the failure is not ‘‘caused’’ by the branch of processing that led to the initial failure, it is caused by the grammar itself with the original input, and would have had occurred even if no decisions had been taken before. So the failure address is switched to the current address.
- Unification succeeds and at least one feature from the grammar is added to the total FD. Forward processing starts again. Any future failure can be caused by these new changes, and cannot therefore be attributed to the old initial failure. So the failure address is also switched to the current address.

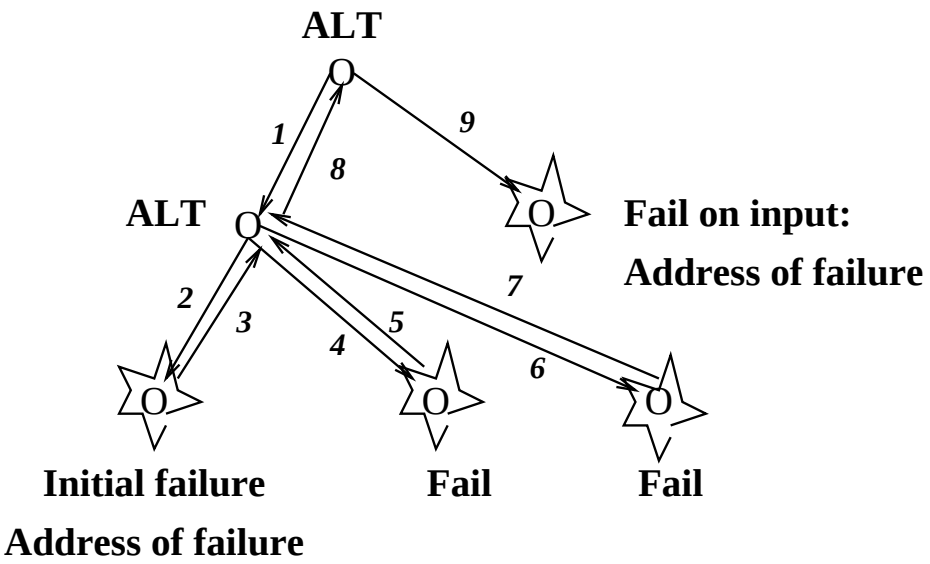


Figure 4-11: Determination of the address of failure

There is one more complication with the use of the address of failure as a source of information on what decision caused the failure: as already mentioned in Sect.3.3.6, a given location in the total FD can be accessed through several paths from the top of the total FD (since an FD is not a tree but can be an arbitrary graph). So several distinct paths, with different labels can identify the location where the failure occurred. The intelligent backtracking component must decide which path should be matched against the bk-class specifications. The convention used here is the same one used to disambiguate the up-arrow notation presented in Sect.3.3.6: the path used to identify the address of failure is the path written in the grammar on the feature that caused the failure. The motivation is that the grammar writer uses the path that is the most meaningful to access features, and that incidental conflations along this path would not be as informative for backtracking purposes. Another approach could have been to use all the path addresses that identify a location and match them against all bk-class specifications. But this approach would have imposed too much overhead on the regular backtracking mechanism.

The bk-class mechanism works well in practice. For the example of Fig.4-6, I have measured the number of backtracking points required to generate different clauses conveying the same core content. The following table summarizes these measurements.[33]

Input / Output	w/o bk-class	w/ bk-class
No floating constraints: *The DN beat the BC*	110	110
Manner in the verb: *The DN edged the BC*	110	110
Manner as adverbial: *The DN narrowly beat the BC*	>10,000	214
AO in the verb: *The DN stunned the BC*	112	112
AO as adjective: *The hapless DN beat the BC*	1,623	239

[33]In this table, *DN* abbreviates *Denver Nuggets* and *BC* abbreviates *Boston Celtics*.

AO as adverbial: *The DN surprisingly beat the BC*	>100,000	268
AO & manner together: *The hapless DN edged the BC*	1,178	246

The number of backtracking points required to generate each example clause is listed with and without `bk-class`. The numbers for the first clause, which does not include any floating constraints, give an indication of the size of the grammar. It roughly corresponds to the number of unretracted decisions made by the grammar. It is the optimal number of backtracking points that a search control regime can obtain for the given input with this grammar. Without `bk-class`, the wide variation in number of backtracking points among the examples indicates the exponential nature of the blind search which floating constraints impose on the standard control regime. In contrast, with `bk-class`, the variation in number of backtracking points remains within a factor of three among all the examples.

4.1.5. Goal Delaying

The dependency-directed mechanism implemented in FUF with `bk-class` complements a general top-down control regime to make the processing of floating constraints efficient. The performance penalty imposed by a floating constraint depends on the number of sites in the syntactic structure where it can be realized. For example, the AO constraint can be realized at three levels and it may require the unifier to re-traverse the grammar three times until it finds a site to convey the AO constraint. Each floating constraint can be characterized by its range of possible attachment nodes. In general, it would be desirable to delay the attachment until it is proven compatible with the other constraints.

To achieve this, I have developed an explicit annotation called `wait` which implements a delay in FUF. It is similar to Naish's implementation of goal delaying for PROLOG [Naish 85]. In FUF, a `wait` annotation freezes the choice of a branch in a disjunction until values for a given set of paths in the total FD are available.

The idea is to freeze a decision until enough information is gathered by the rest of the unification process to make the decision efficiently. The annotation is:

```
(alt (:wait (<path_1> ...)) ...)
```

When the unifier meets such a disjunction, it first checks if all the $path_i$ have a value which is instantiated. If they are all instantiated, it proceeds as usual. If they are not, then the disjunction is put on hold (frozen) and the unification proceeds with the other constraints in the grammar. Whenever a toplevel disjunction is entered, the unifier checks if one of the frozen disjunctions can be thawed. The stack of frozen disjunctions is called the *agenda* and is searched in first-in first-out manner. If the grammar is completely traversed and only frozen decisions remain, an arbitrary frozen alt on the agenda is forced.

The main benefit of `wait` is that it allows ordering the decisions dynamically based on what the input contains. In this section, I first present an application of `wait` in the grammar to deal with ellipsis, illustrating the use of `wait` alone. I then come back to the case of floating constraints and illustrate how `wait` can be used in combination with `bk-class` to improve the handling of floating constraints. Finally, I explain how `wait` interacts with the overall flow of control and specifically the constituent traversal strategy.

4.1.5.1. Wait and Subject Ellipsis

The following example illustrates a use of the `wait` construct which demonstrates that `wait` is a more general delaying mechanism than `bk-class`. The context is in the processing of conjunction and the task is to decide whether to perform an ellipsis of subject, as in: *John ate and left* as opposed to *John ate and John left.* The problem is that in the input, *John* is not identified as a subject (which is a syntactic function), but as an agent (the semantic role). The form of ellipsis I am considering only applies to constituents filling the same syntactic function, regardless of their thematic role. Thus, for example, in *John gave a lot and was given little*, *John* is subject in both clauses, but it is *agent* in the first one and *beneficiary* in the second.

When the grammar is traversed, the order in which constituents are unified is shown in Fig.4-12. A conjunction of clauses is made up of the constituents `constituent1` and `constituent2`[34], each one being a clause.

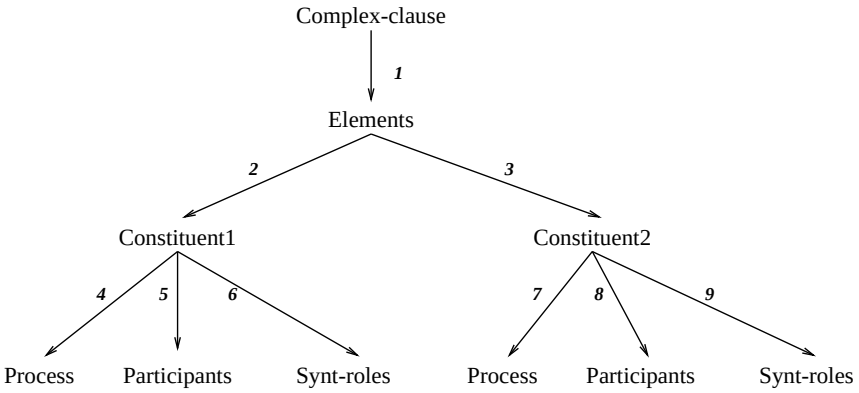


Figure 4-12: Order of constituent processing in a clause conjunction

The mapping between participants (the semantic roles) and syntactic roles is performed by the clause grammar, at the level of `constituent1` and `constituent2`. But the decision to perform ellipsis must be made at the clause-complex level, which is the only level where the grammar knows that the clauses are conjoined and has access to all conjuncts. So the problem is that the clause-complex grammar must make a decision based on information that will be provided by the processing of later constituents.

The solution is to delay the ellipsis decision until each clause has performed the transitivity mapping of participants to syntactic roles. This is implemented using the `wait` annotation as shown in Fig.4-13. Note that if the input already specifies which constituent is the subject in each conjunct, then the decision can be evaluated right away. It is only delayed when needed, and decisions are indeed ordered dynamically. The ellipsis is done by adding a gap feature to the subject in the second constituent. Gap is interpreted by the linearizer which will translate a gaped constituent into an empty string.

```
;; In conjunction: do subject ellipsis?
;; --> John ate and left.
;; The ellipsis is done by adding a GAP feature to the second subject.
;; Wait until subjects for both conjuncts have been mapped.
;; This alt does NOT determine what the subjects should be.
(alt subject-ellipsis (:wait ({^ constituent1 synt-roles subject}
                              {^ constituent2 synt-roles subject}))
  (((cat clause)
    ({^ constituent1 synt-roles subject index}
     {^5 constituent2 synt-roles subject index})
    ({^ constituent2 synt-roles subject gap} yes)
    (subject-ellipsis yes))
   ((subject-ellipsis no)))))
```

Figure 4-13: A grammar fragment for subject ellipsis

The alt subject-ellipsis is only evaluated when both `{constituent1 synt-roles subject}` and `{constituent2 synt-roles subject}` are instantiated. When they are, the first branch checks if the semantic index of both subjects unify and if so adds a `(gap yes)` feature to the second subject. Note that in a similar case, before the `wait` mechanism was implemented, an interesting ‘‘bug’’ occurred: the input to generate *John eats and*

[34]These are actually pointers to the {car} and {cdr car} paths of the elements list.

Mary is hungry produced the sentence *John eats and Mary hungry*. The culprit was found to be a missing `wait` annotation in the alt performing verb-ellipsis. Verb-ellipsis checks if the verbs in the two clauses are the same, and when they are, gaps the second one. Unfortunately, the verb of the second clause is only determined later, when the copula is found appropriate to express the attributive relation between *Mary* and *hungry*. So, in the buggy version, the bidirectional unification mechanism actually instantiated the verb of the second clause to be `eat` and gaped it. Later, the part of the grammar determining the verb for an attributive clause only uses the verb *to be* if the verb is not already specified. Since the verb had already been bound to *to eat*, the grammar assumed it was a correct verb to express the attributive relation, and no failure was detected. To solve this problem, the delaying annotation offered a natural way to express the necessary dependencies between decisions.

4.1.5.2. Wait and Floating Constraints

`Wait` can also be used to deal with floating constraints. For the floating constraint example described above, Fig.4-14 shows which annotations are added to the grammar to delay the processing of floating constraints until enough information is gathered to decide where they should be attached.

```
( ...
 ((sem-cat action)
  (alt (index on concept)
    ;; Map the concept game-result to a verb
    (((concept win)
      (alt win-lex (:bk-class (AO manner))
        (:wait ({^ AO conveyed} {^ manner conveyed}))
        (;; The winner's rating is poor
         (({AO} ((concept c-rating) (carrier {winner})
                 (orientation -) (conveyed yes)))
          (lex ((alt ("stun" "surprise"))))
          ({adverb} none))

         ;; The victory is narrow
         (({manner} ((concept c-tight) (conveyed yes)))
          (lex ((alt ("edge" "nip"))))
          ({adverb} none))

         ;; Default neutral verbs
         ((lex ((alt ("beat" "defeat" "down"))))))))
       ... )))
 ...
 (verb ((alt ...))) ...
 (AO ((alt ...))) ...
 (manner ((alt manner-adverbial (:bk-class manner)
           (:wait ({^ manner conveyed}))
           (;; Has been realized by other means - nothing to add
            ((conveyed any))

            ;; Map manner to an adverbial adjunct
            ;; and mark that manner has been realized
            (({adverb} ((synt-cat adverb) (concept {^ ^ concept})))
             (conveyed adverb))))))
  ... )
```

Figure 4-14: Handling floating constraints with wait

Flow of control works as follows: FUF first checks whether the predicate `win` can be mapped to a verb. Since there is a choice between verbs carrying different connotations and neutral verbs, the decision is delayed by virtue of the `wait` annotation: neither `{AO conveyed}` nor `{manner conveyed}` are instantiated at this point. So the whole `win-lex` alt is put on the agenda. Next, because the verb is not instantiated, many of the subsequent decisions are also delayed (all those having to do with complements in the clause depend on the subcategorization frame of the verb, which is not yet available). Then comes the processing of the two other floating constraints, `AO` and `manner`. Both of these are also delayed, since `conveyed` is not instantiated. The situation at the end is, therefore, a sort of deadlock: no disjunction has forced the decision and all other decisions are blocked as a consequence.

This is when the *determination* stage of the unification algorithm plays a role again. Recall from Sect.3.3.2 that unification proceeds in two stages: first the entire grammar is traversed, making choices in all undelayed disjunctions; then all delayed decisions that were stacked on the agenda are processed. This includes `any` constraints and disjunctions blocked by a `wait` annotation. At determination time, the *determination policy* specifies what is done with delayed constraints: they can be either forced or kept. The agenda policy is set by the grammar writer depending on the architecture of the generator. In most cases, it is set to `force`. When the policy is `force`, delayed constraints are forced in the order in which they have been stored in the agenda (first-in first-out). But there is also an option to keep the agenda. This is useful when two grammars are used in succession, in a pipeline architecture. For example, a first grammar performs lexicalization and a second grammar performs grammaticalization. The delayed constraints of the first grammar can be kept and forced only at the end of the second grammar unification. This second option is discussed in more detail below, in Sect.4.1.6 (p.102).

Thus, in general, the agenda is checked at determination time, and the alt that was first delayed is forced. The advantage of delaying all floating constraints is that all structural decisions are first taken, as much of the syntactic tree as possible is built without taking any floating constraint into consideration. Then all floating constraints are evaluated one after the other. Therefore, order of decision is dynamically ordered so that floating constraints are processed as late as possible and all processed close to one another. Thanks to this property, backtracking due to the interaction between floating constraints is reduced.

4.1.5.3. Wait vs. Bk-class

While backtracking can be minimized by the use of `wait`, it cannot be avoided entirely. In this paragraph, I explain how `wait` and `bk-class` interact in a synergistic way to limit backtracking when complex constraint interactions must be processed.

In the floating constraints example, when all structural processes that do not depend on the floating constraints have been processed, a deadlock situation is created. To break the deadlock, FUF selects one of the constraints arbitrarily.[35] Intuitively, FUF must choose non-deterministically between the verb, adjective and adverb ways of realizing the AO floating constraint. This non-deterministic choice can lead to backtracking. In this case, a combination of `bk-class` and `wait` is necessary to minimize backtracking, as explained here.

Thus, in the example, when determination starts (after the entire grammar has been traversed and all remaining decisions have been frozen), the choice of verb is forced. The verb *stun* is chosen (it is the first choice available and it satisfies the AO constraint). Since the verb is available, many of the blocked alts that depended on the verb are now unblocked and are processed again, in the same order as they would have been in the original pass through the grammar. In particular, the AO alt is also unblocked since the `{AO conveyed}` path is now instantiated. Finally, determination starts again after all unblocked constraints have been processed. This time, only the `manner` alt is still frozen. When it is unblocked, the grammar realizes that no slot has been left available to express it: adverbs cannot be used because a marked verb is used. So backtracking starts. The main issue in this example is that the choice of the verb plays such a central role in the construction of the syntactic structure, that very little of the syntactic tree could be built until the verb was chosen. When it was selected, all the structural decisions depending on the verb are evaluated, and as a consequence, the `manner` alt is evaluated very late with respect to the verb choice. In this case thus, `wait` alone would not prevent expansive backtracking, and `bk-class` is still needed.

In general, the use of `wait` improves the situation over the use of `bk-class` alone. The following table summarizes the measurements for the same cases presented earlier using `bk-class`:

Input / Output	bk-class alone	bk-class and wait
No floating constraints: *The DN beat the BC*	110	118
Manner in the verb: *The DN edged the BC*	110	118

[35] In fact, it is not completely arbitrarily, since FUF uses the order in which the alts have been traversed in the grammar, but it is a blind non-deterministic choice anyway.

Manner as adverbial: *The DN narrowly beat the BC*	214	121
AO in the verb: *The DN stunned the BC*	112	118
AO as adjective: *The hapless DN beat the BC*	239	289
AO as adverbial: *The DN surprisingly beat the BC*	268	122
AO & manner together: *The hapless DN edged the BC*	246	295

These numbers show that in general, the `wait` annotations prevent the grammar from being traversed twice to generate an adverb: the first time without conveying the manner, the second time to add the adverb. Instead, the choice of adding an adverb is delayed, and at the end, if it is possible and necessary the adverb is added without causing any backtracking at all. Thus in lines 3 and 6 of the table, the use of `wait` cuts the required number of backtracking points in two.

But in cases where several floating constraints compete (for example when the AO constraint can be realized either as an adjective or as an adverbial or when both manner and AO are present), the `wait` annotations do not help more than the `bk-class` mechanism alone. Luckily, they do not harm either, introducing only a small overhead of approximately 10%. So the combination of `bk-class` with `wait` is in general very constructive.

4.1.5.4. Wait and Constituent Traversal

The delaying of disjunctions interacts in a complex manner with the way the constituent structure is traversed. Recall that the traversal proceeds as follows: the toplevel constituent is unified with the grammar, then at the end of the unification (before determination time), the `cset` of the constituent is identified, and every descendant specified in the `cset` is traversed, in order. The structure is then traversed breadth first.

There are two ways to specify a `cset` in a constituent in FUF: implicitly and explicitly. The `cset` is explicitly specified if there is a complete `cset` feature found in the constituent. Otherwise it is implicit. The implicit `cset` is found by using two heuristics: first it is assumed that any path mentioned in a pattern is also a constituent; second, any feature which contains a sub-feature of the form `(cat xx)` is also assumed to be a constituent.

To override these heuristics, the grammar writer can use *incremental cset specifications*. There are indeed two types of `cset` specifications: complete and incremental. A complete specification is of the form `(cset (c1 ... cn))` and exhaustively identifies all the subconstituents of the current constituent. An incremental specification is of the form:

`(cset (+` a_1 `...` a_m`) (-` d_1 `...` d_p`))`

An incremental `cset` specifies that all the a_is should be added to the implicit `cset` and all the d_js should be removed from it. The possibility of specifying the constituent structure incrementally is a great convenience practically since it allows the grammar writer to give partial information on constituency in different regions of the grammar.

Incremental cset specifications may affect delayed disjunctions. The following example illustrates this situation:

```
((alt (:wait {^ manner conveyed})
  (((manner ((conveyed any)))
   ((manner ((conveyed adverb)
             (cat adverb) ...))
    (cset ((+ manner))))))))
```

In this case, the decision to express the manner constraint by using an adverb is delayed, leaving a chance for the verb to express it as a connotation. Thus, in the first pass through the grammar, this alt is not evaluated. At the end of the processing of the clause constituent, the cset is determined. The implicit cset is computed and all the incremental cset specifications are added on it. At this point, the `manner` feature is not identified as a constituent,

since there is no cat specified for it and no incremental cset has added it explicitly. The constituent structure is then traversed. At the end of this traversal, the delayed alts on the agenda are processed. The manner alt is thus unblocked, and the second branch is eventually selected. An adverb constituent is then added at the clause level. When the alt is completely processed, this adverb constituent should be unified with the adverb grammar as any subconstituent would be, if it had been processed without being delayed. Unfortunately, the constituent structure was already computed before the delayed alt was unblocked. And the adverb was not part of the cset at this time. So the adverb is not added to the final constituent structure of the clause.

I have extended the determination process to re-check the constituent structure whenever an alt has been unblocked during determination, thus allowing new constituents to be added by delayed alts without restriction. The determination process performs the following task: first, a delayed alt is selected (the first one that was put on the agenda is chosen) and its evaluation is forced. Next, unification starts again to evaluate all the delayed alts which have been unblocked by the evaluation of the forced alt. At the end of this unification step, determination starts again. This loop continues until the agenda is empty. At this point, the constituent structure is recomputed and retraversed from the top of the total FD down, in breadth-first order. All constituents which have already been unified are skipped, all new constituents identified by the recomputation of the `csets` at the end of the agenda evaluation are unified again. After this unification step, determination starts again from scratch (since the evaluation of the new constituents may have added delayed alts on the agenda), until the agenda is empty and all constituents have been unified.

This top-down traversal in several passes integrates nicely the regular top-down control regime with the benefits of the dynamic re-ordering of constraints provided by `wait`. Note that this traversal in several passes does not translate in reduced efficiency: it is only triggered when needed (that is, when decisions had to be delayed), and the traversal/recomputation of the constituent structure is a very efficient operation. As measured, efficiency is overall drastically improved with `wait`.

4.1.6. Conditional Evaluation

The last control tool implemented in FUF is conditional evaluation, implemented with the annotations `ignore-when`, `ignore-unless` and `ignore-after`. The idea here is to just ignore certain decisions when either there is enough information already in the input, there is not enough information or there are not enough resources. The 3 cases correspond to the annotations:

```
(alt (:ignore-when <path> ...) ...)
(alt (:ignore-unless <path> ...) ...)
(alt (:ignore-after <number>) ...)
```

`Ignore-when` is triggered when the paths are already instantiated. It is used when the input already contains information and the grammar does not have to re-derive it.

`Ignore-unless` is triggered when a path is not instantiated. It is used when the input does not contain enough information at all, and the grammar can just choose an arbitrary default.

`Ignore-after` is triggered after a certain number of backtracking points have been consumed. The annotation indicates that the decision encoded by the disjunction is a detail refinement that is not necessary to the completion of the unification, but would just add to its appropriateness or value.

One characteristics of these annotations is that their evaluation may depend on the order in which evaluation proceeds. Since this order is not known to the grammar writer, their use can be delicate. To prevent unpredictable variations, the `ignore` annotations should be used in conjunction with `wait`, since `wait` establishes constraints on when a decision is evaluated. Therefore a `wait` annotation has priority over an `ignore` when both occur in the same alt. Adding a `wait` annotation can often ensure that the `ignore` annotation will only be considered when it is relevant.

`Ignore-when` is used, for example, in the grammar for NPs to declare that the whole NP branch does not need to be traversed when the NP constituent contains a gap (the feature `(gap yes)` when it appears in a constituent means that the constituent is linearized as an empty string). When an NP is gapped, indeed, there is no need to compute its

internal structure. Similarly, in the determiner grammar, `ignore-when` is used to declare that the whole determiner branch can be ignored when the determiner constituent already contains an instantiated `lex` feature. In general, the determiner is a sequence of closed class items which are derived by the grammar from semantic and pragmatic features. If the `lex` feature already specifies which string should be used, then this derivation does not need to be computed. This allows the grammar to take full advantage of a mixed input, containing both semantic and linguistic constraints.

`Ignore-unless` is used, for example, in the grammar for conjunctions to declare that the ellipsis decisions only apply to constituents of category clause. The declaration is shown in Fig.4-15.

```
;; In conjunction: do subject ellipsis?
;; --> John ate and left.
;; This alt is only evaluated when {constituent1 cat}
;; is a specialization of clause.
(alt subject-ellipsis (:wait ({^ constituent1 synt-roles subject}
                              {^ constituent2 synt-roles subject}))
       (:ignore-unless (({^ constituent1 cat} #(under clause))))
  (((cat clause)
    ({^ constituent1 synt-roles subject index}
     {^5 constituent2 synt-roles subject index})
    ({^ constituent2 synt-roles subject gap} yes)
    (subject-ellipsis yes))
   ((subject-ellipsis no)))))
```

Figure 4-15: A grammar fragment for ellipsis with ignore

`Ignore-after` is currently not used in the grammar. It is implemented in FUF but I have not had a need for it in the current work. It would be useful in a situation where resources are limited. Consider again the example of the close basketball game result that was realized above as *the hapless Denver Nuggets edged the Boston Celtics 101-99*. The number of backtracking points required to produce this ‘‘efficient’’ realization is 295. In contrast, a less optimized realization, which leaves out one of the floating constraints (*e.g., the Denver Nuggets surprisingly beat the Boston Celtics 101-99*), only requires 122 backtracking points. The generation of an additional simple sentence like *the game was close* would require 60 backtracking points. So, the part of the grammar which realizes the manner floating constraint can be annotated with an `ignore-after` declaration that indicates that when the manner constraint conflicts with some other constraints in the grammar and that the conflict resolution might require a complex search process, it can just be ignored. The annotation required is:

```
(manner ((alt manner-adverbial (:bk-class manner)
          (:wait ({^ manner conveyed}))
          (:ignore-after 100)
          ...)))
```

This annotation means that if this decision is delayed and then evaluated after more than 100 backtracking points have been used, then it can be simply ignored. As a result, the manner constraint of the input is not satisfied by the clause being constructed, and the `any` constraint requiring it to be satisfied remains on the agenda. This configuration would fail under normal conditions. This is when a different agenda policy becomes useful. Recall that at determination time the agenda is checked. Normally if any delayed construct is found in the agenda, it is forced. This is the behavior when the agenda policy is `force`. Another option is to use an agenda policy of `keep`: delayed constraints are simply kept on the agenda (in the form of FDs), and are passed to a subsequent process. In the example, the ‘‘repair’’ process would have to replan a way to express the manner constraint that could not go through the first clause and generate a semantic form for an additional clause. While all the elements to implement such a mechanism are implemented in FUF, I have not experimented with this type of resource-limited processing. Some issues need to be addressed to make this scheme practical: in particular, the number of backtracking points is not a good measure of the effort that has been spent on generation. But the mechanism seems promising.

4.1.7. Summary: Control in FUF and Efficiency

I have presented in this section a collection of control tools which make FUF a practical generation system, able to handle a wide variety of input constraint configurations efficiently. In particular, I have identified a class of input constraints, called floating constraints, which require complex search during generation. Handling floating constraints is necessary for lexical choice because lexical choice involves mapping a conceptual structure onto a non-isomorphic linguistic structure. Handling configurations where several conceptual elements are merged into a single lexical entry or where the same conceptual element can be realized at different syntactic levels is crucial to developing a flexible lexical chooser, and especially important to deal with argumentative evaluations. The control tools presented in this section are well adapted to minimize the search space for this particular class of constraints.

The general approach to control in FUF is to add annotations to the grammar at disjunctions that express information on how decisions depend on each other and on the instantiation of features in the total FD. These annotations are used by the unifier to determine the most appropriate order of decision making. The simplest annotation, `index`, improves efficiency uniformly by a factor of 1/5 to 1/4. More complex annotations, `bk-class` and `wait`, prevent the unifier from entering costly exponential searches to deal with conflicting goals.

The overall control strategy is a top-down traversal of the constituent structure. This top-down regime avoids non-termination in left-recursive contexts by using the `given` meta-variable to ensure that linguistic constituents are only added when they actually cover existing semantic elements. In addition, goal freezing (with `wait`) can also be used to avoid non-termination in left-recursive rules.

The control annotation mechanisms presented in this section improve FUGs' efficiency while preserving their desirable properties - declarativeness and bidirectional constraint satisfaction. Annotations can be read declaratively as statements of dependency between decisions in the grammar and classes of constraints in the input or between decisions in the grammar and feature instantiation in the total FD.

Using these annotations is not always easy for the grammar writer since it requires thinking about the control strategy of the unifier - the same drawback as for PROLOG's `cut` mechanism. Determining where in the grammar annotations such as `wait` and `bk-class` need to be added is facilitated when distinguishing between floating and structural constraints. For a given floating constraint, the grammar writer must identify all the syntactic sites where it can be realized and annotate the choice points in the grammar that attach these sites to the syntactic tree with `bk-class` or `wait`. The reasoning involved when adding these annotations is in a sense orthogonal to the reasoning involved when developing the grammar in the first place: when writing the grammar, choice points are introduced to capture the different types of functions that can be realized by a specific linguistic device (for example, a clause can express a certain list of process types); in contrast, annotations are introduced to express the different linguistic devices that can express the same function (for example, enumerate all the syntactic sites which can realize a manner or an argumentative evaluation and annotate them). This approach to the grammar is often illuminating and enriches the understanding of linguistic resources from a different perspective than standard grammars do.

In addition, FUF's annotations are optional, and can be added only when needed to optimize a grammar when performance becomes an issue.

A few issues in the implementation of these annotations require further optimization. Specifically, for all delayed constraints one must determine how often the agenda must be checked to determine whether a blocked alt is now unblocked and which alts of the agenda should be checked.

Optimally, it would be desirable that whenever a feature is instantiated the delayed alts that depend on it be notified. But this would impose too much overhead on the general feature instantiation mechanism. Practically, the first issue is the most important: in the current implementation, the agenda is checked before any disjunction is entered in the grammar and at determination time. This can impose a high overhead on the whole unification process when many disjunctions are on the agenda. A form of ''granularity'' control system, which would control how often the agenda is checked would allow to decrease this overhead.

Finally, in many cases, when backtracking occurs across constituents (*e.g.*, from the determiner of a noun group up to the clause), the changes that are made high up in the clause do not affect the decisions that were made in the

sub-constituents. In such a case, a sort of chart-based approach would be desirable so that when the sub-constituents are re-traversed after backtracking they do not need to be re-unified, and decisions made once are never re-evaluated. A sort of *memoization* of the unification at the level of constituents would certainly enhance efficiency [Norvig 91].[36]

4.2. Types and Usability Issues

When developing SURGE, I have found that the original functional unification formalism is not well suited for the expression of simple, yet very common, taxonomic relations. The traditional way to implement such relations in FUG is verbose, inefficient and unreadable. Since many such relations are expressed in the grammar, it is important to optimize their expression. It is also impossible to express completeness constraints on descriptions - as a consequence, it is sometimes difficult to check the validity of input descriptions and to detect ill-formed input FDs. Finally, certain constraints are just too complex to be expressed in FUGs, or else in a very complicated form. For example, arithmetic relations cannot be expressed in FUGs without resorting to a very low-level axiomatic. If such constraints were to be expressed anyway in FUG, the size of the grammar and its readability would be affected.

In this section, I present three forms of typing extending the functional unification formalism to address these limitations of the original FUG formalism. By solving these problems, typing also enhances the maintainability of large grammars, by making them shorter, easier to read and document.

I first introduce the notion of typed features. It allows the definition of a structure over the primitive symbols used in the grammar. The unifier can take advantage of this structure in a manner similar to [Ait-Kaci 84]. I then introduce the notion of typed constituents and the FSET construct. It allows the declaration of explicit constraints on the set of admissible paths in functional descriptions. Finally, we introduce procedural typing, allowing the expression of constraints that are beyond the expressive power of FUGs or that could be cumbersome to express in pure FUG.

Most work in computational linguistics using a unification-based formalism (*e.g.*, [Uszkoreit 86; Karttunen 86; Kay 79; Kaplan & Bresnan 82]) does not make use of explicit typing. In [Ait-Kaci 84], Ait-Kaci introduced Ψ-terms, which are very similar to feature structures, and introduced the use of type inheritance in unification. Ψ-terms were intended to be general-purpose programming constructs. I base the extension for typed features on this work but I add the notion of typed constituents and the ability to express completeness constraints. I also integrate the idea of typing with the particulars of FUGs (notion of constituent, NONE, ANY and CSET constructs) and show the relevance of typing for linguistic applications. Similar work also derived from [Ait-Kaci 84] is also being developed in the grammatical framework of HPSG [Pollard & Sag 87]. I compare later in this section the approach presented here to this work.

4.2.1. Typed Features

4.2.1.1. A Limitation of FUGs: No Structure over the Set of Values

In functional unification, the set of constants *C* that specifies the allowed values of attributes has no structure. It is a flat collection of symbols with no relations among them. All constraints between symbols must be expressed in the grammar. In linguistics, however, grammars assume a rich structure between properties: some groups of features are mutually exclusive; some features are only defined in the context of other features.

Let us consider a fragment of grammar describing noun-phrases (NPs) (cf Fig.4-16) using the systemic notation given in [Winograd 83]. Systemic networks such as this one, as discussed in Sect.3.1.2.2, encode the choices that need to be made to produce a complex linguistic entity. They indicate how features can be combined or whether

[36]Memoization is a technique that turns a function into a memo-function. A memo-function is a function which caches pairs of input/output forms. When an input which has already been processed is evaluated again, the memo-function just returns the value from its cache without recomputing it.

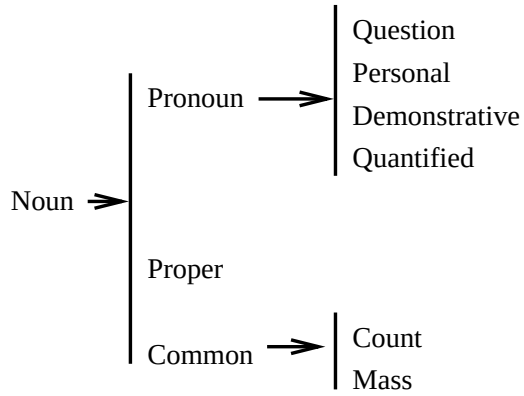


Figure 4-16: A system for NPs

features are inconsistent with other combinations. The configuration illustrated by this fragment is typical, and occurs very often in grammars: the schema indicates that a noun can be either a pronoun, a proper noun or a common noun. Note that these three features are mutually exclusive. Note also that the choice between the features `{question, personal, demonstrative, quantified}` is relevant only when the feature `pronoun` is selected. This system, therefore, forbids combinations of the type `{pronoun, proper}` and `{common, personal}`.

The traditional technique for expressing these constraints in a FUG is to define a label for each non terminal symbol in the system. The resulting grammar is shown in Fig.4-17. This grammar is, however, incorrect, as it allows combinations of the type `((noun proper) (pronoun question))` or even worse `((noun proper) (pronoun zouzou))`. Because unification is similar to union of feature sets, a feature `(pronoun question)` in the input would simply get added to the output. In order to enforce the correct constraints, it is, therefore, necessary to use the meta-FD NONE in the grammar (which prevents the addition of unwanted features) as shown in Fig.4-18.

```
((cat noun)
 (alt (((noun pronoun)
        (pronoun ((alt (question personal demonstrative quantified)))))
       ((noun proper))
       ((noun common)
        (common ((alt (count mass))))))))
```

Figure 4-17: A faulty FUG for the NP system

There are two problems with this corrected FUG implementation. First, both the input FD describing a pronoun and the grammar are redundant and longer than needed. Second, the branches of the alternations in the grammar are interdependent: the branch for pronouns must represent the fact that common nouns can be sub-categorized and what the other classes of nouns are. This interdependence prevents any modularity: if a branch is added to an alternation, all other branches need to be modified. It is also an inefficient mechanism as the number of pairs processed during unification is $O(n^d)$ for a taxonomy of depth d with an average of n branches at each level.

4.2.1.2. A Solution: Typed Features

The problem is that FUGs do not gracefully implement mutual exclusion and hierarchical relations. The system of nouns is a typical taxonomic relation. The deeper the taxonomy, the more problems there are expressing it using traditional FUGs. In the current version of the SURGE grammar, I have found that the type of taxonomic relations discussed above occurs more than 80 times. Thus, with a grammar containing approximately 500 disjunctions, 16% of the disjunctions would have to be devoted to an implementation of this type of relations.

```
((alt (((noun pronoun)
        (common NONE)
        (pronoun ((alt (question personal demonstrative quantified)))))
       ((noun proper)
        (pronoun NONE)
        (common NONE))
       ((noun common)
        (pronoun NONE)
        (common ((alt (count mass))))))))
```

The input FD describing a personal pronoun is then:
`((cat noun) (noun pronoun) (pronoun personal))`

Figure 4-18: A correct FUG for the NP system

```
(define-type noun (pronoun proper common))
(define-type pronoun (personal-pronoun question-pronoun
                      demonstrative-pronoun quantified-pronoun))
(define-type common (count-noun mass-noun))
```

The grammar becomes:

```
((cat noun)
 (alt (((cat pronoun)
        (cat ((alt (question-pronoun personal-pronoun
                    demonstrative-pronoun quantified-pronoun)))))
       ((cat proper))
       ((cat common)
        (cat ((alt (count-noun mass-noun))))))))
```

And the input: `((cat personal-pronoun))`

Figure 4-19: Using typed features

I propose extracting hierarchical information from the FUG and expressing it as a constraint over the symbols used. The solution is to define a subsumption relation over the set of constants *C*. One way to define this order is to define types of symbols, as illustrated in Figure 4-19. This is similar to Ψ-terms defined in [Ait-Kaci 84]. Once types and a subsumption relation are defined, the unification algorithm must be modified. The atoms *X* and *Y* can be unified if they are equal OR if one subsumes the other. The result is the most specific of *X* and *Y*.

With this new definition of unification, taking advantage of the structure over constants, the grammar and the input become much smaller and more readable as shown in Fig.4-19. There is no need to introduce artificial labels. The input FD describing a pronoun is a simple `((cat personal-pronoun))` instead of the redundant chain down the hierarchy `((cat noun) (noun pronoun) (pronoun personal))`. Because values can now share the same label CAT, mutual exclusion is enforced without adding any pair `[l:NONE]`.[37] Note that it is now possible to have several pairs `[a:`v_i`]` in an FD F, but that the phrase ''the `a` of F'' is still non-ambiguous: it refers to the most specific of the v_i. Finally, the fact that there is a taxonomy is explicitly stated in the type definition section whereas it used to be buried in the code of the FUG. This taxonomy is used to document the grammar and to check the validity of input FDs, thus improving the robustness of the grammar. Figure 4-20 shows some of the types actually used in the FUF grammar. The total type system currently defines 83 hierarchical relations over 250 symbols defining syntactic categories, semantic categories, types of participants and events, tenses and moods. The deepest hierarchy has five levels. The encoding of these relations as types instead of the sort of imperfect disjunctions shown above has reduced the size of SURGE by approximately 15% in addition to the perspicuity it brings.

[37] In this example, the grammar could be a simple flat alternation ((cat ((alt (noun pronoun personal-pronoun ... common mass-noun count-noun))))), but this expression would hide the structure of the grammar.

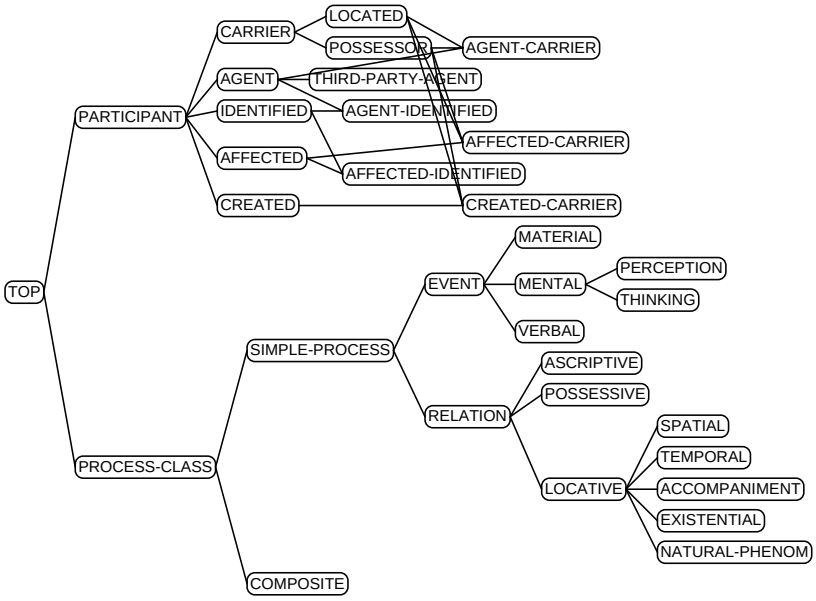


Figure 4-20: Types used in the transitivity region of the clause grammar

4.2.2. Typed Constituents: the FSET Construct

A natural extension of the notion of typed features is to type constituents: typing a feature restricts its possible values; typing a constituent restricts the possible features it can have.

```
Type declarations:
(define-constituent determiner
  (definite distance demonstrative possessive))

Input FD describing a determiner:
(determiner ((definite yes)
             (distance far)
             (demonstrative no)
             (possessive no)))
```

Figure 4-21: A typed constituent

Figure 4-21 illustrates the idea. The `define constituent` statement allows only the four given features to appear under the constituent `determiner`. This statement declares what the grammar knows about determiners. `Define constituent` is a completeness constraint as defined in LFGs [Kaplan & Bresnan 82]; it says what the grammar needs in order to consider a constituent complete. Without this construct, FDs can only express partial information. Note that expressing such a constraint (a limit on the arity of a constituent) is impossible in the traditional FU formalism. It would be the equivalent of putting a NONE in the attribute field of a pair as in NONE:NONE.

Without FSET:

```
(define-constituent inherent-roles
  (agent affected benef carrier attribute processor phenomenon))

((cat clause)
 (alt (((process ((type action)))
        (inherent-roles ((carrier NONE)
                         (attribute NONE)
                         (processor NONE)
                         (phenomenon NONE))))
       ((process ((type attributive)))
        (inherent-roles ((agent NONE)
                         (affected NONE)
                         (benef NONE)
                         (processor NONE)
                         (phenomenon NONE))))
       ((process ((type mental)))
        (inherent-roles ((agent NONE)
                         (affected NONE)
                         (benef NONE)
                         (carrier NONE)
                         (attribute NONE)))))))
```

With FSET:

```
((cat clause)
 (alt (((process ((type action)))
        (inherent-roles ((FSET (agent affected benef)))))
       ((process ((type attributive)))
        (inherent-roles ((FSET (carrier attribute)))))
       ((process ((type mental)))
        (inherent-roles ((FSET (processor phenomenon))))))))
```

Figure 4-22: The FSET Construct

In general, the set of features that are allowed under a certain constituent depends on the value of another feature. Figure 4-22 illustrates the problem. The fragment of grammar shown defines what inherent roles are defined for different types of processes (it follows the classification provided in [Halliday 85]). But the grammar also needs to enforce the constraint that the set of inherent roles is ‘‘closed’’ when the process has a particular value: for an action, the inherent roles are agent, affected and benef *and nothing else*. This constraint cannot be expressed by the standard FUG formalism. A `define constituent` makes it possible, but nonetheless not very efficient: the set of possible features under the constituent `inherent-roles` depends on the value of the feature `process type`. The first part of Fig.4-22 shows how the correct constraint can be implemented with `define constituent` only: all the roles that are not defined for the process-type must be excluded. Note that the problems are very similar to those encountered on the pronoun system: explosion of NONE branches, interdependent branches, long and inefficient grammar.

To address this issue, I introduce the construct FSET (feature set). FSET specifies the complete set of legal features at a given level of an FD. FSET adds constraints on the definition of the domain of admissible paths Δ. The syntax is the same as CSET. Note that all the features specified in FSET do not need to appear in an FD: only a subset of those can appear. For example, to define the class of middle verbs (*e.g., to shine* which accepts only an `affected` as inherent role and no `agent`), the following statement can be unified with the fragment of grammar given in Fig.4-22:

```
((process ((type action)
           (lex "shine")))
 (voice-class middle)
 (inherent-roles ((FSET (affected)))))
```

The feature `(FSET (affected))` can be unified with `(FSET (agent affected benef))` and the result is `(FSET (affected))`.

Typing constituents is necessary, for example, to implement the theoretical claim made in LFG that the number of

syntactic functions is limited. It also has practical advantages. The first advantage is good documentation of the grammar. Typing also allows checking the validity of inputs as defined by the type declarations.

Another advantage is that it can be used to define more efficient data-structures to represent FDs. As suggested by the definition of FDs, two types of data-structures can be used to internally represent FDs: a flat list of equations (which is more appropriate for a language like PROLOG) and a structured representation (which is more natural for a language like LISP). When all constituents are typed, it becomes possible to use arrays or hash-tables to store FDs in LISP, which is much more efficient. I am currently investigating alternative internal representations for FDs (cf. [Pereira 85; Karttunen 85; Boyer 88; Hirsh 88] for discussions of data-structures and compilation of FUGs).

In [Emele & Zajac 90], a slightly different approach to constituent typing is presented, which is very close to the typing system used in HPSG [Pollard & Sag 87, Chap.8]. In this approach, **all** FDs must have a type, and the type hierarchy is completely declared statically. Unification includes a step of type inference, where an FD is recognized as a member of the most specific type it matches. The main difference between the FUF's FSET approach and this more radical typing system is that in FUF, constituent typing is optional and dynamic while in the HPSG approach it is obligatory and static: two FDs unify only if their types are compatible as stated in a static type hierarchy. In [Emele & Zajac 90], only FDs which are instances of types which have been statically pre-defined can be represented in the formalism (as if all FDs in FUF had an FSET). In FUF, FSET is optional, and is added by the grammar only at the levels where there is a reason to restrict the number of features. In general, it seems that a type system of the HPSG typed unification can be implemented as a grammar using FSETs.

4.2.3. Procedural Typing

FUF also implements a third notion of type in unification: procedural types correspond to user-defined data-structures that are unified by special-purpose unification methods. The unification method describes how elements of the type fit in a partial order structure. Typed features are extensionally described by partial order relations. With procedural types, the partial order is intensionally described by a LISP function.

Procedural types, therefore, allow the grammar to integrate complex objects that could not be described by standard FDs alone. The original FUG formalism actually included special cases of procedural unification without generalizing it: `pattern` with the pattern unification method enforcing ordering constraints and `cset` with the cset unification method checking for set equality.

I have found other uses for procedural types in FUF's grammar. For example, I now present the `tpattern` (temporal pattern) attribute, used in SURGE to account for the semantics of tenses.[38] Temporal patterns express temporal relations between the time of speech, the time of the action being referred to in a clause and a set of reference times. Each of these times is viewed as an interval and these intervals are related by one of Allen's binary constraints [Allen 83]. Such a constraint configuration defines the temporal semantics of the clause, in a manner derived from [Reichenbach 47] and [Moens & Steedman 88].

In turn, each grammatical tense is represented by a network of constraints on speech time, event time and some reference times. The tense analysis provided in [Halliday 85, Sect.6.3] which describes 36 distinct tenses in English is used in the grammar. The semantics of each tense is represented by a temporal pattern as shown in Fig.4-23.

In this figure, the semantics of the simple past is expressed by the simple relation that the event-time referred to precedes the speech time. A more complex tense (tense 13 in Halliday's classification, or ''past in future in past'') is also shown and involves three reference times in addition to the speech time. The configuration expressed by this `tpattern` corresponds to the diagram shown in Fig.4-24. Note that there is no constraint between `:rt0` and `:st` and `:rt2` and `:st`.

To generate the tense of a clause, the input must contain a temporal pattern, describing actual relations between

[38]The tpattern grammar and unification procedure have been designed in collaboration with David Rabinowitz.

```
;; Tense descriptions in the grammar
(def-alt tense-selection (:index tense)

  ;; Tense 1: past
  ;; I took the bus.
  ;; Simple tense: :st for speech time, :et for event-time
  ((tense past)
   (tpattern ((:et :precedes :st))))

  ...

  ;; tense 13
  ;; I was going to have taken the bus.
  ;; two reference times: :rt0 and :rt1.
  ((tense tense-13)
   (tpattern ((:et  :precedes :st)
              (:et  :precedes :rt0)
              (:rt0 :precedes :rt1))))

  ...)
```

Figure 4-23: Temporal patterns corresponding to English tenses

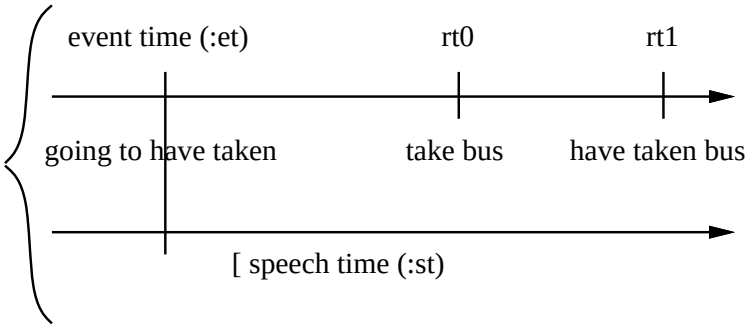


Figure 4-24: Temporal configuration

definite time intervals, as shown in Fig.4-25. The generator then tries to find the most specific temporal pattern compatible with the input pattern among the patterns describing the available tenses of English. This operation is essentially an operation of unification between temporal patterns.

```
;; Input FD for John had run.
((cat clause)
 (process ((type material) (effective no) (lex "run")))
 (tpattern ((:rt0 :precedes :st)
            (:et :precedes :rt0)))
 (participants ((agent ((cat proper) (lex "John"))))))
```

Figure 4-25: Input FD containing a temporal pattern

The problem I met when trying to implement this mechanism for tense selection was to first express temporal patterns as FDs and second make sure that the unification mechanism of FUGs would be consistent with the

semantics of the temporal relations. In turned out that the expression of such temporal patterns as FDs is quite difficult: a temporal pattern is an unordered list of binary relations. A binary relation can be expressed by an FD like:

```
((relation :precedes)
 (arg1 :rt1)
 (arg2 :st))
```

But the representation of unordered lists is quite difficult. The list representation described in Sect.3.4 is ordered. The unification of two temporal patterns requires random access to any of the elements. So a pure FUG implementation would have required many uses of the append and member procedures discussed in Sect.3.4. Instead, the tpattern procedural type was defined. Typed unification allows one to extend the expressiveness of the formalism while keeping the original flavor. Thus, a specialized `tpattern`-unifier was developed whose role is to unify two `tpattern`s together and return the most specific `tpattern` compatible with both when possible, or fail if the input arguments are not compatible.

To maintain the clean semantics of FUGs, certain limitations were imposed on the use of procedural types:

1. Procedurally typed objects are always considered as leaves in an FD: that is, no matter how complex is the object, the unifier does not know how to traverse it from the outside. It is viewed as a black box. There is no notion of path *within* the object.
2. Typed objects can only be unified with objects declared of the same type.
3. It is the responsibility of the user to make sure that a specific unification method actually implements a real partial-order.

```
;; Unification of 2 numbers is the max: order is the total order of
;; arithmetic  (which is also a partial order!)

(defun unify-numbers (n1 n2 &optional path)
  (max n1 n2))

(define-procedural-type 'num 'unify-numbers :syntax 'numberp)

LISP> (u '((num 1)) '((num 2)))
((num 2))

LISP> (u '((num 1)) '((num 0)))
((num 1))

;; Unification of 2 lists is the list with the more elements.
;; That defines a (probably not very useful) total order on lists.
(defun unify-lists (l1 l2 &optional path)
  (if (> (length l1) (length l2))
      l1
    l2))

(define-procedural-type 'list 'unify-lists :syntax 'sequencep)

LISP> (u '((list (1 2 3))) '((list (1 2 3 4))))
((list (1 2 3 4)))

LISP> (u '((list (1 2))) '((list (a b c d))))
((list (a b c d)))
```

Figure 4-26: Simple examples of procedural types

Figure 4-26 shows trivial (read: useless) examples of how procedural types can be used. Procedural types are defined by a name and a special unification method, optionally a syntax checker function can be declared. The unification method must define a partial order over the elements of the type.

```
(DEFINE-PROCEDURAL-TYPE <name> <function>
                        :syntax <checker> :copier <copier>)
```

Declares <name> to be a special attribute, whose value can only be interpreted by <function>, the type unifier.

The unification function must be *deterministic* (no backtracking is allowed) and must be a real ''unification'' procedure; that is, the type must be a lattice (or partial order). Practically, the unification function must satisfy the following constraints:

- FUNCTION must be a function of 3 arguments: the values to unify and the path where the result is to be located in the total fd. It must return :fail if unification fails, otherwise, it must return a valid object of type TYPE.
- FUNCTION must be such that NIL is always acceptable as an argument and is always neutral, *i.e.*, (FUNCTION x nil) = x.
- FUNCTION must be such that (FUNCTION x x) = x

CHECKER must be a function of 1 argument: It must return either True if the object is a syntactically correct element of TYPE, otherwise, it must return 2 values: NIL and a string describing the correct syntax of TYPE.

COPIER must be a function of 1 argument: it must copy an object of TYPE that has no cons in common with its argument. By default, copy-tree is used.

NOTE: (<COPIER> x) = (<FUNCTION> x nil)

For example, the tpattern-unifier is a complex function that is aware of the semantics of Allen's temporal relations and can determine when two constraint networks represented by tpatterns are compatible. It works by converting tpatterns to a canonic form which is ''saturated'' (that is, all relations that can be inferred from the input network are explicitly represented) and ordered (a custom order is defined over the relations and the restricted set of symbols that can represent the nodes, *e.g.*, :st or :rt0).

In summary, procedural types offer a natural extension to the typed feature mechanism by allowing the partial order between leaves of FDs to be expressed in intension (by a function) rather than in extension (by a set of subsumption relations). It increases the expressiveness of the formalism by allowing the description of complex data structures that would be too cumbersome to express in pure FUGs and their manipulation in terms of unification.

4.2.4. Summary: Typing in FUF

Functional Descriptions are built from two components: a set *C* of primitives and a set *L* of labels. In the original FUG formalism, all structuring of FDs is done using strings of labels. I have shown in this section that there is much to be gained by delegating some of the structuring to a set of primitives. The set *C* is no longer a flat set of symbols, but is viewed as a richly structured world. The idea of typed-unification is not new [Ait-Kaci 84], but I have integrated it for the first time in the context of FUGs and have shown its linguistic relevance: it allows the encoding of common taxonomic relations among linguistic features and categories. I have also introduced the FSET construct, not previously used in unification, endowing FUGs with the capacity to represent and reason about complete information in certain situations.

The structure of *C* can be used as a meta-description of the grammar: the type declarations specify what the grammar knows, and are used to check input FDs. It allows the writing of much more concise grammars, which perform more efficiently. It is a great resource for documenting the grammar.

Finally, in practical systems, some data structures that are too complex to be described as FDs need to be represented and manipulated by the grammar. Procedural typing allows the grammar writer to describe such data structures as typed objects along with a special-purpose unification procedure defining a partial order over the set of such objects.

The three varieties of typing offered in FUF increase the expressiveness of the formalism while keeping its semantic

properties (typing minimally affects the definition of unification, which remains a monotonic operation). They also improve the usability of FUF for large grammars by making them more concise and easier to read and maintain.

4.3. Modularity in FUF

During the development of FUF, the grammar has grown in two ways: first, the grammar itself has become larger, as the syntactic coverage extended; second, FUF has grown to cover more aspects of the generation process - starting with simple syntactic coverage to later include lexicalization and the processing of non-linguistic factors like topoi. These two dimensions have led to the need of two distinct ways of controlling the size of a FUG. I present first in this section two tools which allow the development of modular architectures around FUF: `external` allows FUGs to interact smoothly with external non-unification processes which are required to develop a complete generation system and parameterized csets allow the construction of pipelines of FUGs. Finally, I introduce a simple syntax to define FUGs in a modular way

4.3.1. External: Modularity and Interleaving[39]

Content realization consists of mapping a semantic input structure onto a syntactic tree. In addition to the constraints present in the input, this process is constrained by a heterogeneous set of factors. Such factors are surveyed in [Matthiessen 91] and [Robin 90]. They include:

- grammar rules
- a conceptual lexicon specifying the mapping between domain concepts and lexical items
- a grammatical dictionary providing the special grammatical properties of lexical items
- a collocation dictionary providing the restrictions on lexical co-occurrences
- a discourse model keeping track of the structure of the text as it is generated
- a domain knowledge base representing the encyclopedic context of generation
- a user-model representing the interpersonal context of generation

These sources vary along several dimensions:

- *Structure*: the grammar rules and the conceptual lexicon express *structural* constraints. They specify a transformation from one regular structure to another. Other sources like the collocation dictionary express inherently non-structural constraints [Halliday 76b, p.73].
- *Portability*: the grammar rules, the grammatical dictionary [Cumming 86] and to some extent the collocation dictionary [Smadja 91b] are domain-independent. The other sources are highly domain-dependent.
- *Dynamism*: the discourse model is inherently dynamic, changing from one sentence to the next. In some applications [Dale 88], this is also the case for the domain model and the user-model. The other sources are static.

How can these knowledge sources be combined?

One approach would be to integrate all these constraints into a single FUG. In addition to being non-modular and thus hindering portability, this approach is impractical for dynamic constraints: being a monotonic process, unification is inadequate to update dynamic models as generation unfolds.

A modular architecture is therefore preferable. The structural constraint sources - conceptual lexicon and grammar

[39]This section describes work done in collaboration with Jacques Robin. It is derived from [Elhadad & Robin 92].

rules - can readily be implemented as a FUG as they are well handled by FUF's top-down regime. During unification, this backbone FUG needs to communicate with the other sources when necessary. What is needed is a mechanism allowing constraints that lie outside of both the input FD and the FUG to be taken into account at any point during the unification process.

I introduce the `external` construct to address this need. When FUF encounters a feature of the form `(a #(external F))` it performs the following operation:

1. Unification is suspended.
2. The external function `F` - a LISP function returning a sub-FD - is evaluated.
3. The value returned by `F` becomes the new value of the attribute `a` in the total FD.
4. Unification resumes where it was suspended with the updated total FD.

Therefore `External` allows the dynamic expansion of a FUG at unification-time.

To illustrate the use of the `external` construct in FUF, consider again the task of generating the sentence: *The hapless Denver Nuggets edged the Celtics 101-99.* As explained in Sect.4.1.3, the verb *to edge* in this sentence not only realizes the predicate element of the semantic input but also the manner constraint. This floating constraint provides a qualitative evaluation of the basketball game reported by the sentence. Such a qualitative evaluation does not depend only on the final score of the game[40] but on other quantitative factors as well, such as the number of lead changes in the final minutes or the largest lead by either team at any point during the game. Several such quantitative facts about a game are abstracted and conflated with the semantic predicate by verbs like *to edge*, *to hammer*, *to outlast* or *to rally past.* Choosing among these verbs requires deciding which combination of quantitative facts is abstracted by the manner connotation of each verb.

In Sect.4.1.3, I assumed that this decision was performed by the content planner building the FUG semantic input as illustrated by the presence of the `(concept c-tight)` feature in the input of Fig.4-14. Performing such a mapping, however, requires knowledge of the existing lexical resources available in a given sublanguage. The fact that a given combination of quantitative data about a basketball game can be compactly expressed in English by describing the game as *tight* is *linguistic knowledge*. It should therefore be located in the lexicon portion of the FUG. Using `external` allows FUF to enforce this separation between linguistic and conceptual knowledge. In this case, the semantic input to the FUG no longer needs to provide a pre-linguistic specification of the manner. It needs only indicate that one of the sentence's communicative goals is to express the manner. Instead of the feature: `(manner ((sem-cat quality) (concept c-tight)))`, the semantic input just contains the feature: `(manner any)`. The lexico-grammar is now in charge of choosing a lexical item to appropriately qualify the game. To perform this choice, it must access the description of the game in the encyclopedic knowledge base. Figure 4-27 illustrates the architecture suggested by this observation. Figure 4-28 shows a fragment of a lexicon where the `external` construct implements an example of such query.

Unification of the semantic input with the `win-lex alt` of this lexicon fragment triggers calls to external functions. Each of these functions queries the knowledge base for quantitative data and returns an FD containing corresponding qualitative features. For example, the function `get-lead-changes` shown in Fig.4-28 enriches the total FD with the feature `lead-changes`. These external functions connect the FUG with the knowledge base. Within the body of an external function, one can access any feature in the total FD by specifying its path prefixed by `@`. In Fig.4-28, note how this notation is used in the `get-lead-changes` function to retrieve information from the knowledge base about the particular token given in the semantic input.[41]

After the external functions return, the features added to the total FD are used for choosing a verb conveying the manner connotation appropriate to this particular game. For example, the verb *to edge* is preferred when a combination of features signals that (1) there was no overtime (2) no team built a big lead and (3) there were many lead changes. As Robin, through a corpus analysis of basketball reports, has identified more than 100 different

[40] In which case it would be redundant with the quantitative expression of the score in the sentence.

[41] Recall that the semantic input is part of the total FD at any point during unification.

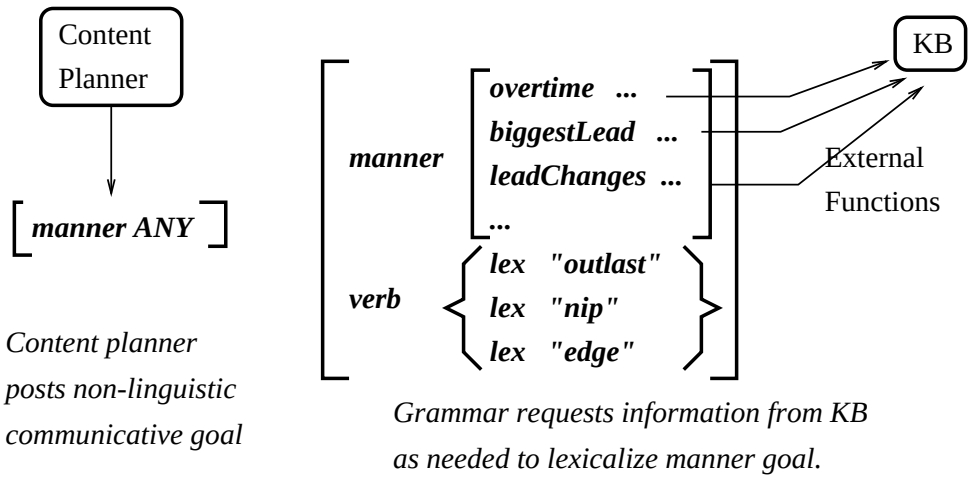


Figure 4-27: An architecture using external functions

An external query function:

```
(defun get-lead-changes ()
  (let ((lead-change-num (get-role-value (get-token @{token})
                                         'lead-change-num)))
    (cond ((> lead-change-num 15) '((lead-changes numerous)))
          ((and (> lead-change-num 5) (> 15 lead-change-num))
           '((lead-changes average)))
          ((> 5 lead-change-num) '((lead-changes few))))))
```

Backbone lexico-grammar with `external` *constructs*:

```
(...
 ((sem-cat action)
  (concept c-win)
  (alt win-lex (:bk-class (AO manner))
    ((({manner} any)

      ;; Knowledge base query for information discriminating
      ;; among manner-conveying verbs.
      ({manner} ((overtime #(external #'get-overtime))
                 (biggest-lead #(external #'get-biggest-lead))
                 (lead-changes #(external #'get-lead-changes))
                 ...))
      (alt win-and-manner-lex

        (;; victory was obtained in overtime after many lead changes
         (({manner} ((overtime yes) (lead-changes numerous)))
          (lex "outlast"))

         ;; victory was close and obtained in regulation
         (({manner} ((overtime no) (biggest-lead small)
                     (lead-changes numerous)))
          (alt (((lex "edge") (lex "nip")))))
         ...)))))))
```

Figure 4-28: Accessing an external knowledge source from the lexicon

verbs in the basketball sublanguage to express the victory of a team, many features are required to discriminate between them [Robin 92b]. However, in a given situation, only a few features will actually be needed. Having the content planner systematically retrieve all the knowledge base information necessary to discriminate between all the words of the lexicon would thus be computationally wasteful.

In addition, the set of features required to discriminate between words depends on the part of speech. For example, there are much fewer adverbs available to convey the manner connotation than there are verbs. Fewer features are therefore required to discriminate among manner adverbs than among verbs conveying a manner connotation. Requiring the content planner to provide the features for all classes of lexical choice in advance would therefore impair modularity between linguistic and conceptual knowledge.

To summarize, the `external` construct enhances FUF in the following ways:

- It provides a co-routine control structure to interact with external processes.
- It enforces an information-hiding principle between different knowledge sources.
- It is a way to fetch constraints lying outside the FUG *on demand*, only when needed by FUF to choose between alternatives.

These different points correspond to needs that have been identified in many generation systems. TELEGRAM [Appelt 85] implemented a mechanism where a FUG and a content planner cooperated to generate referring expressions. The `external` construct is a generalization of this mechanism. With PAULINE, Hovy ([Hovy 88a]) advocated interleaving pervasively content realization with content planning. With `external`, FUF can implement such an interleaving while benefiting from the declarative nature of FUGs. While traversing a systemic linguistic network, PENMAN [Mann 83] accesses its environment by calling functions called *inquiries*. FUF's `external` functions provides a similar facility in the context of FUGs. Finally, DIOGENES, [Nirenburg and Nirenburg 88] uses a blackboard to communicate with and control specialized modules working with their own separate knowledge sources. With `external`, a similar cooperation among specialized modules can be implemented in FUF with the total FD playing the role of the blackboard.

4.3.2. Pipelines of Grammars

One common way of defining the architecture of a generator is by defining a pipeline of successive modules. For example, a text planner produces an input for the text realization component, or a lexical chooser produces an input for the grammar realization component.

This is, for example, the strategy used in the COMET system: the lexical choice module is implemented as a FUG and produces a fully lexicalized input for a purely syntactic FUG. Actually COMET contains a pipeline of three FUGs: COMET is a multimedia explanation generator which produces both text and graphics to assist a user in the maintenance of complex electronic equipment (cf. [Feiner and McKeown 91] for an overview of the whole system and [McKeown et al 90] for a description of the natural language component of COMET). The COMET system includes a media-coordinator which determines which medium is appropriate for each type of information to be conveyed to the user. The COMET system includes a three-part pipeline of FUGs to connect the media-coordinator (implemented as a FUG) to the lexical chooser and to the grammar. The relevant part of the architecture is shown in Fig.4-29.

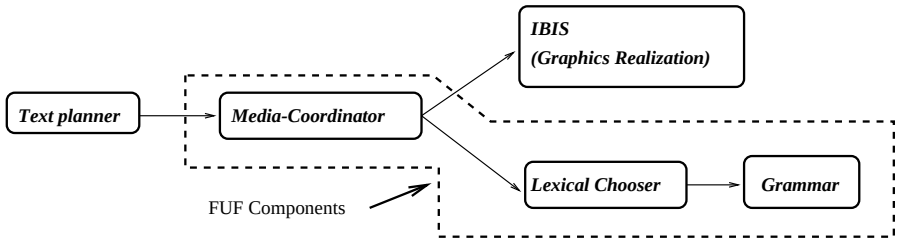


Figure 4-29: FUF pipeline in the COMET architecture

This is also the architecture used in ADVISOR II, where lexical choice and syntactic realization are implemented using two successive grammars written in FUF. The advantage of a pipeline architecture is that when order of decision is clearly defined, each component can concentrate on its own task, remaining simple. The complexity of the overall

architecture is the sum of the elements of the pipeline instead of its product if backtracking were to be allowed. The main limitation of pipeline architectures is precisely that backtracking is not allowed from one component to the previous one, and decisions made in an early module cannot be revised by a later one. In any case, pipelines of FUGs are common and I have developed tools in FUF to ease the development of such pipelines.

The main issue when connecting several FUGs in a pipeline is that the output of one stage must be an acceptable input for the next stage. There are features, however, which are only used locally for the processing at one level, and should not interfere with the later stages. Among these, the features describing the constituent structure of the FD are the most important: `cset` and `cat` are the two constants used extensively by the unifier to determine how to traverse the FD (`cat` is used during the implicit `cset` computation). The problem is that the constituent structure is different at each level of the pipeline. For example, in the COMET system, the media-coordinator considers the input as a pure semantic structure and the `csets` identify at each level the semantic arguments of semantic heads. At the lexical choice stage, the traversal uses a different view of what constitutes the structure: lexical heads identify their own lexical dependents. If the `cset` of the media coordinator remain in the FD when the lexical chooser runs, the second unification would not work.

To avoid this confusion, FUF has turned the structure-denoting features from constants into parameters. Thus, both `cset` and `cat` are arguments to the unifier. The unifier function is called as in:

```
(unify fd lex-grammar :cat 'lex-cat :cset 'lex-cset)
(unify fd sem-grammar :cat 'sem-cat :cset 'sem-cset)
```

Different features are used at each stage, and can coexist in the same total FD, without interfering:

```
((sem-cat relation)
 (lex-cat clause)
 (sem-cset (predicate agent affected))
 (lex-cset (verb subject object)))
```

With this approach, several constituent structures can be described and processed concurrently in the same total FD.

4.3.3. Modular Organization of Grammars: Def-alt and Def-conj

A large FUG, like many other large computer programs, becomes difficult to read and maintain when its size increases. The size of a FUG is best measured by counting the number of disjunctions it contains. The depth of embedding of the disjunctions is also an indication of the complexity. The number of branches in the disjunctive normal form of the grammar accounts for this embedding but is in general not very informative. For example, the grammar of FUF contains 580 disjunctions and has a disjunctive normal size of 10^{29}. Another more conventional measure of the complexity of the grammar viewed as a program, is the number of lines of code: 9,500 lines for a total of 290,000 characters. By all measures, such a grammar is a large program. So the well-known problems of size management in software engineering become an issue for large FUGs: ease of maintenance, readability, modularity etc.

A FUG is a large FD containing many alternations. At the top-level, a FUG is conventionally made up of a large alternation where each branch describes a different category. The pattern is:

```
((alt (((cat clause) ...)
       ((cat noun-group) ...)
       ((cat verb-group) ...)
       ...)))
```

Each branch is an embedded FD containing in turn many disjunctions. For example, the top-level clause grammar is made up of the following alternations:

```
((cat clause)
 (alt mood ...)
 (alt transitivity ...)
 (alt voice ...)
 (alt circumstantial ...)
 (alt displaced-constituent ...)
 ...)
```

Each one of these alternation in turn is a large FD, with many embedded alternations.

I have designed a very simple mechanism to allow the grammar writer to develop such large grammars in a modular way. The key aspect is to allow the grammar writer to abstract away from the details of an FD by giving it a name. Named FDs can then be used anywhere a regular FD is allowed. It turned out that instead of using named FDs, named features were much more convenient. The new syntax distinguishes between two types of named features: named disjunctions and named conjunctions defined by `def-alt` and `def-conj` respectively. Named features can then be used in any FD as regular features; that is, instead of a pair, an FD can contain a named feature. The syntax is illustrated in Fig.4-30.

```
(def-alt number (:index number)
   (((number singular) ...)
    ((number plural) ...)))

(def-conj agreement-3-sing-masc
   (number singular)
   (person third)
   (gender masculine))
```

Using named feature

```
((alt (((cat noun-group)
        (:! number)
        ...))))

((head ((lex "man")))
 (:& agreement-3-sing-masc))
```

Expanded form

```
((alt (((cat noun-group)
        (alt (:index number)
          (((number singular) ...)
           ((number plural) ...)))
          ...))))

((head ((lex "man")))
 (number singular)
 (person third)
 (gender masculine))
```

Figure 4-30: Using def-alt and def-conj

A reference to a named disjunction has the form `(:! name)`; `(:& name)` is used for named conjunction. These forms can appear anywhere a pair could appear in an FD. A named conjunction works like a bundle of features which get spliced in the embedding FD when it is referenced. A named disjunction accepts all the annotations that an alt can carry (control and tracing annotations). The def-alt and def-conj mechanism works as a macro mechanism for grammars.

This simple syntactic tool allows the use of abstraction in the development of FUGs: by naming parts and hiding levels of details, the structure of the grammar becomes more apparent. Named features can be re-used in different contexts (several places in the grammar and/or in several grammars). Practically, it becomes possible to work separately on a module of the grammar without affecting the other parts; several people can work together on the same grammar; single modules can be re-loaded and re-defined without requiring re-loading the whole grammar. The standard benefits of modularity in programming languages apply to the full extent.

Interestingly, the new syntax highlights the similarity between FUGs and the systems of systemic linguistic. I have developed a tool that draws a graphical map of the grammar by displaying the tree of the named features with their dependents (function `draw-grammar`). A high-level map of the grammar used in Advisor II is shown in Fig.4-31. Note how similar this map is to a system in systemic linguistic. The new syntax has made this level of organization clearly visible in the grammar without requiring any change to the formalism.

4.3.4. Summary: Modularity in FUF

I have argued in this section for the need for modularity in generation. Modularity is necessary to manage large grammars and to split the generation process among several specialized processes. This is especially important for lexical choice for two reasons: first, lexical choice and syntactic realization can be implemented as two distinct

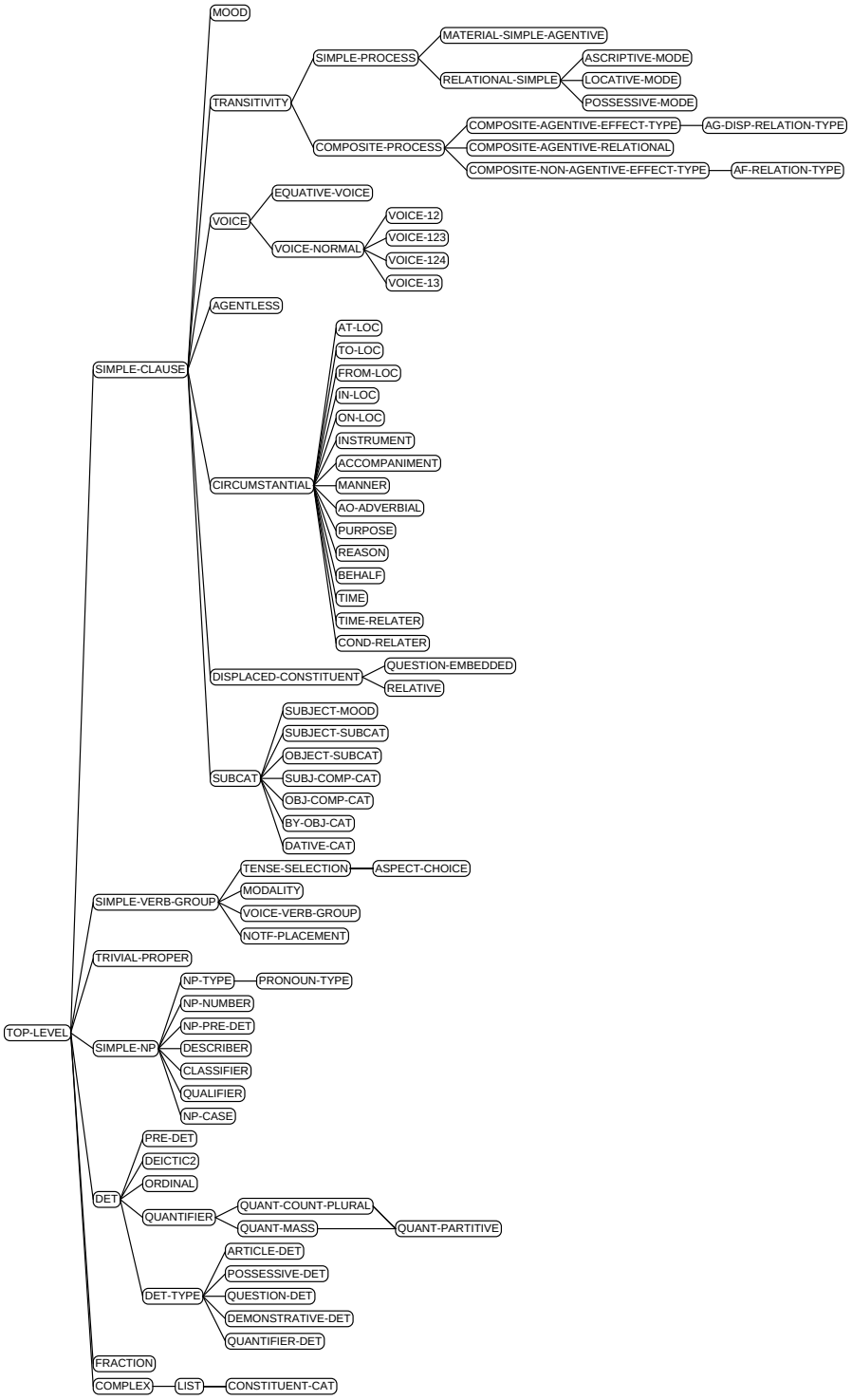


Figure 4-31: Map of the grammar

grammars, and second, in a modular architecture, the lexical choice grammar can interact with external knowledge sources, such as the knowledge base and the user model, using the `external` construct. This interaction relieves the content determination from determining in advance all the features that the lexical chooser may need.

I have presented tools supporting modularity in FUF: `external` allows the interleaving of unification with the external, non-unification based processes required to implement a complete generation system; parameterized `csets` allow the construction of pipelines of FUGs with distinct structures being built and manipulated at each stage; and named features are a syntactic extension which allows the grammar writer to define abstractions over feature structures and reuse them in an easy way.

4.4. Summary: FUF as a Practical Generation System

The extensions to the FUG formalism presented in this chapter turn FUF into a complete practical generation environment. I have extended the formalism to address limitations with the original FUG in three areas: efficiency with a collection of powerful control facilities adapted to the needs of a generation system; usability with an extended type facility; and modularity with facilities allowing the development of modular grammars and the integration of FUF in a modular architecture.

Thanks to these new facilities, FUF is able to represent and handle the complex constraints required to perform lexical choice and particularly to efficiently handle interacting constraints. In particular, I have identified the class of floating constraints, which is critical to the implementation of lexical choice, and shown how the new control tools presented in this section minimize the search required to handle floating constraints.

FUF has been used to develop large systems and the facilities presented in this chapter have evolved from extensive usage to answer the needs of a varied community of users. Most of the techniques and tools presented here originate from techniques developed in programming languages, in particular logic programming. The main contribution of FUF as a formalism is to integrate a selective combination of techniques into the original FUG formalism which for the most part do not affect its semantics and specifically address the needs of the generation task. The main contribution of FUF as an implemented system is to provide a robust, flexible and extensible programming environment specialized to the task of generation. At the basic level, usage of FUF only requires understanding of the simple and elegant FUG formalism. When required, the more advanced features can be used to address the limitations of existing grammars as they become visible. Experience with FUF has indeed confirmed that the system is easy to learn yet powerful enough to grow with experienced users.

Chapter 5
SURGE: A Large Portable Syntactic Realization Grammar

This chapter presents the SURGE syntactic realization grammar. SURGE is a large portable grammar implemented in FUF[42]. The role of a syntactic realization grammar in the generation process is to process the output of the lexical chooser and transform it into a syntactically complete specification. This syntactic specification is passed to a linearizer/morphology module which translates it into English text. The place of SURGE in the generator's architecture is depicted in Fig.5-1.

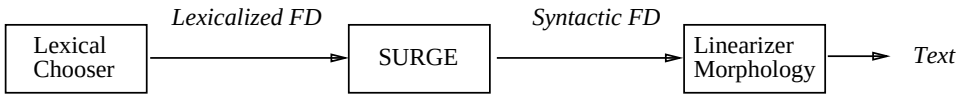


Figure 5-1: Place of the SURGE in the generation process

I present SURGE at this point for two main reasons. First, SURGE is portable and domain-independent, and together with the FUF system, it provides a reusable surface generator. As such, the package FUF/SURGE is integrated, and the three chapters presenting these two components form a coherent description of this portable package. The second reason is that, before describing lexical choice in the next chapter, this chapter defines the output that the lexical chooser must produce. The role of the syntactic realization component is pretty well defined. That of the lexical chooser is less clear, and there is no agreement among researchers as to what exactly is the lexical choice task. For that reason, a precise delimitation of the task of the lexical chooser in the ADVISOR II system starts with a definition of what its output must be. This chapter provides precisely such a definition.

Thus, the main function of this chapter is to describe the set of features that the lexical chooser must provide to drive the SURGE grammar. It does not cover all the capabilities of SURGE, but only those that are accounted for by the lexical chooser of ADVISOR II.

SURGE is the result of five years of intensive experimentation in grammar writing for surface generation. This chapter does not have any new syntactic theory to contribute. It presents rather an *implementation* of a large computational grammar. The encoding of different syntactic models in the FUF formalism has often yielded a clear understanding of the syntactic issues. An important goal of this chapter is to demonstrate the ease with which the FUF formalism can express some of the most complex syntactic phenomena. SURGE is mainly inspired by systemic linguistics. The main sources used when developing the grammar are [Halliday 85] for the overall organization of the grammar and a large part of the clause grammar; [Fawcett 87] for the transitivity system; [Winograd 83, App.B]; and overall [Quirk *et al* 72] for the most comprehensive presentation of English grammar and its invaluable attention to detail. [Quirk *et al* 72] is the most descriptive work of the lot, not being overtly bound to any formal linguistic school. It therefore provides a sort of theory neutral treasure chest of organized observations. As the grammar increased in coverage and the issue of lexical choice became clearer in my mind, new influences have emerged. Mel'cuk's Meaning-Text Theory (MTT) and the rich description of English surface syntax provided in [Mel'cuk and Pertsov 87] have provided many insights. Pollard and Sag's Head-driven Phrase Structure Grammars (HPSG) have provided the impetus to address syntactically challenging issues such as command and control [Pollard & Sag 87]. I

[42] SURGE stands for Semantic Unification-based Realization Grammar of English

do not include in this chapter an account of how the SURGE analysis has been derived from these diverse sources. Instead, I present SURGE as a coherent grammar.

In the rest of this chapter, I first define in general terms the task of a syntactic realization grammar. I then provide a detailed description of the clause grammar, the NP grammar and the determiner sequence grammar. As mentioned above, this is not a description of the full SURGE grammar. It only defines the features accounted for by the lexical chooser. SURGE is still rapidly evolving, and new features are being added continuously. SURGE has been distributed to more than 30 research sites worldwide and I am aware of seven research teams actively using SURGE on twelve distinct projects, as of Summer of 1992. This wide distribution has helped (1) debug the grammar; (2) extend its coverage and robustness; (3) insure its portability and domain independence. This description is therefore to be interpreted as a snapshot of the current state of the grammar.

5.1. Role of a Syntactic Realization Grammar

The function of a syntactic realization grammar is to map a lexicalized input onto a syntactic output. The output is a description of an English sentence[43] containing all the morphological features and ordering constraints necessary so that a linearizer and morphology module can translate it into syntactically correct English. For example, verbs must be encoded in the output with at least the following features: a string representing the root of the verb, tense, person and number. These are the features required by the morphology module to produce a fully inflected and conjugated form for the verb. All words must be present in the output of the syntactic realization grammar, including closed-class items such as determiners and prepositions.

The input to the syntactic realization grammar contains a semantico-linguistic encoding of the content to be realized. In addition, all open-class lexical items are specified. The input in Fig.5-2 illustrates the form of features used in SURGE. Input1 is a simple FD describing the clause *I am working*. In this clause, the only open-class item is the verb *work*. The other semantic element is the pronoun *I*, described in terms of two features: person and number. The structure of the clause is described in semantic terms: this is a material process with one participant, the agent. In general, a goal of a syntactic realization grammar is to provide a level of abstraction from the syntax, allowing the specification of the input in the most semantic terms possible.

```
(def-test input1
  "I am working."
  ((cat clause)
   (process ((type material) (lex "work") (effective no)))
   (partic ((agent ((cat personal-pronoun)
                    (person first) (number singular)))))
   (tense present-progressive)))
```

Figure 5-2: A simple input to SURGE

The output which SURGE produces from Input1 is shown in Fig.5-3 (this FD has been edited to make it more readable). As usual with a functional unification grammar, this FD is an enriched version of the input FD. I discuss in the following paragraphs the function of SURGE by explaining the function of most of the enriched features in Fig.5-3.

Briefly, the main functions of a syntactic realization grammar are:

- Define syntactic structure.
- Provide ordering constraints among the syntactic constituents of the sentence.
- Propagate agreement features between the words of the sentence, providing input to the morphology module to compute inflections and conjugations.

[43] In this discussion, I will use the term 'sentence' to refer to the output of SURGE mainly for convenience. Keep in mind, however, that SURGE can also output a segment comprising several sentences.

- Select closed-class words.
- Prevent over-generation.
- Provide ways to express possible inputs for as many syntactic forms as possible (coverage).
- Provide defaults for syntactic features.
- Control grammatical paraphrasing
- Perform syntactic inference.

I now develop each of these points.

5.1.1. Construct a Syntactic Structure

The first task of SURGE is to map the semantic structure of the input onto a syntactic structure. The syntactic structure defines constituents, that is, how words are grouped in the sentence. Figure 5-4 shows how the semantic input of the example is enriched by a syntactic structure. In the figure, the original semantic structure is shown with the three bold arrows. The other links indicate the new syntactic structure derived by SURGE. The process semantic constituent is mapped to the main verb of the clause. The verb itself is structured as a be-1 auxiliary followed by the event to encode the progressive form *am working*.

The mapping of the semantic participants onto the syntactic roles is a little more complex: it is performed in two stages. First, the semantic participants are mapped to a level called ‘‘oblique’’, then the oblique frame is mapped to syntactic functions (subject, object, indirect object, subject-complement or object-complement). In the example, the final syntactic function is thus made up of the verb, of category verb-group and the subject, of category NP.

5.1.2. Provide Ordering Constraints

The second task of SURGE is to provide ordering constraints between the constituents, so that the linearizer can output a linear string of words. Ordering constraints are expressed in FUF using the pattern feature. A pattern is placed in the FD describing each syntactic constituent: one at the top level for the clause, one at the verb level and one at the subject level. Patterns are primordial since in English word-order is significant.

5.1.3. Propagate Agreement

The third task of SURGE consists in propagating agreement features between constituents. For example, the subject / verb agreement is encoded in Fig.5-3 by conflating the features person and number between verb and subject, using the following FUF expressions at the verb level:

```
(person {synt-roles subject person})
(number {synt-roles subject number})
```

Similarly, many features in the output FD are passed down from the clause to the verb (tense, polarity), and from the subject NP to its head (*e.g.*, number, case, person, animate). Such feature passing is an explicit encoding in SURGE of phenomena captured by the head feature principle and the foot feature principle in GPSG, for example [Gazdar et al 85].

The agreement features insure that the leaves of the syntactic tree contain the appropriate morphological features required to drive the morphology module and produce properly inflected forms.

```
((cat verbal-clause) (generic-cat clause)
 (process ((type material) (lex "work")
           (process-type material)
           (effective no) (agentive yes)
           (voice active)
           (subcat {oblique})
           (cat simple-verb-group) (generic-cat verb-group)
           (modality none) (tense {tense}) (polarity {polarity})
           (person {synt-roles subject person})
           (number {synt-roles subject number})
           (event ((cat verb) (lex {process lex})
                   (ending present-participle)))
           (be-1 ((tense present) (person {process person})
                  (number {process number}) (ending {process ending})
                  (lex "be") (cat verb)))
           (pattern (dots be-1 dots event dots))))
 (partic ((agent
           ((cat personal-pronoun) (generic-cat np)
            (person first) (number singular)
            (synt-function subject) (case subjective)
            (syntax ((case subjective)
                     (fset (animate gender case definite person
                            partitive number a-an distance countable))
                     (animate yes) (number singular)
                     (person first) (definite yes)))
            (semantics ((fset (index describer qualifier classifier))
                        (index ((fset (concept animate gender person
                                       number denotation countable))
                                (animate yes) (number singular)
                                (person first)))))
            (reference ((fset (type total selective possessive degree
                               interrogative distance quantitative
                               exact orientation evaluative status
                               comparative superlative evaluation))
                        (type specific) (possessive no)
                        (interrogative no) (quantitative no)))
            (pattern (determiner head dots))
            (head ((cat pronoun) (number singular)
                   (animate yes) (pronoun-type personal)
                   (person first) (case subjective)))
            (determiner ((cat article-det) (generic-cat det)
                         (head-cat pronoun)
                         (det ((cat article) (lex "")))))
            (possessive no) (quantitative no) (interrogative no)
            (partitive no) (animate yes) (definite yes)))
          (fset (agent))))
 (tense present-progressive)
 (oblique ((fset (1)) (1 {participants agent})))
 (synt-roles ((fset (subject)) (subject {oblique 1})))
 (focus {subject})
 (verb {process}) (mood declarative)
 (pattern (dots start {^ synt-roles subject} dots verb dots))
 (circumstances ((fset (at-loc to-loc from-loc on-loc in-loc instrument
                        accompaniment manner temporal-background
                        purpose reason behalf time))))
 (dative-move no) (insistence no) (polarity positive))
```

Figure 5-3: Output from SURGE for Input1

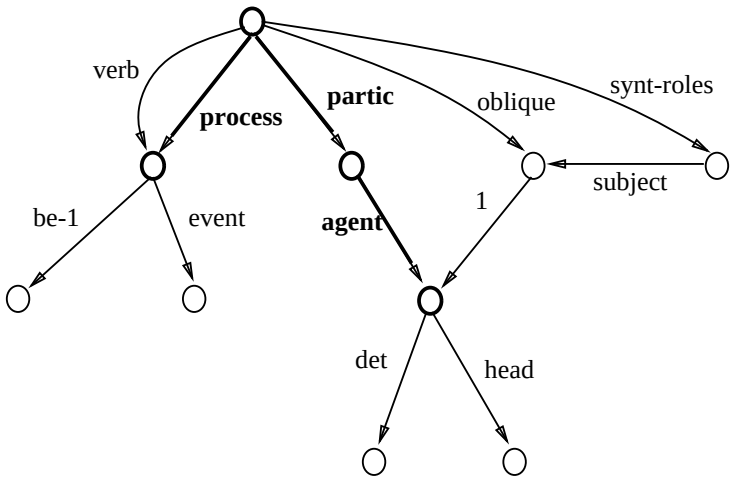


Figure 5-4: Structure built by SURGE for Input1

5.1.4. Select Closed-class Words

The input to SURGE specifies all open-class lexical items. The closed-class items are generated by SURGE. For example, the auxiliary be-1 is included by SURGE to express the progressive form (*I am working*). Other classes of closed-class items generated by SURGE are prepositions introducing adjuncts, based on the semantic function of the adjunct, determiners, pronouns, based on person, number, case and function (interrogative, relative ...).

5.1.5. Prevent Over-generation

SURGE must allow only the generation of syntactically valid sentences. To prevent over-generation, SURGE includes constraints on the categories of fillers for each syntactic function.

5.1.6. Provide Wide Coverage

SURGE must allow a wide variety of syntactic forms to be generated. An overview of SURGE's coverage is provided in the next sections, when the detailed grammars of the clause, NP and determiners are presented.

5.1.7. Provide Defaults for Syntactic Features

The output shown in Fig.5-3 illustrates that the number of syntactic features considered by SURGE is quite large. For example, to build the determiner sequence of an NP, SURGE requires the value of 24 features. SURGE provides defaults for all the syntactic features used in the grammar. Thanks to these defaults, the input does not need to specify the value of all syntactic features and remain compact.[44]

[44]The sentence generated with an empty input, *i.e.*, containing only default features is: *it is.*

5.1.8. Control Grammatical Paraphrasing

For a given semantic input FD, several syntactic forms can be generated. For example, the following alternations provide purely syntactic paraphrases: active/passive voice, dative move, passive with and without agent. The choice between these variants is not always free. It may depend on other syntactic decisions or on pragmatic factors. A syntactic realization grammar must provide features controlling how these variants are selected and select the appropriate form when the choice is not free.

For example, the dative move is constrained by several factors:

(1) John gives a book to Mary.
John gives Mary a book.

(2) John gives it to Mary.
** John gives Mary it.*

(3) To whom does John give a book?
** Whom does John give a book?*

In (1), the alternation is free. In contrast, in (2) and (3), the form with a to-complement must be used, because a pronoun is used in (2) and an object is moved in question position in (3). In such cases, there is no choice. SURGE recognizes these cases and enforces the constraint. When the choice is free, the feature dative-move controls whether the form with *to* or the form with the indirect object is used.[45]

5.1.9. Perform Syntactic Inference

A final function of a syntactic realization grammar is to perform ‘‘syntactic inference’’. By this term, I refer to a process of constraint propagation which forces certain syntactic features to a certain value because of the interaction with other features. For example, consider the following case of control:

SURGE requires the input to be lexicalized by the lexical chooser.

```
Require(Influence: SURGE,
          Influenced: Input,
          State-of-affairs: lexicalize(LC, Input))
```

In this sentence, the verb *Require* has two complements: *Input* and the infinitive clause *to be specified by the lexical chooser*. In addition, there is a control relation which imposes that *Input* also be the subject of the infinitive clause. When generating this sentence from the logical form shown above (in a shortened FD format), SURGE can infer that the clause realizing the `lexicalize(LC, Input)` relation must be put in the passive voice. The reason is that `Input` is the `affected` role of the `lexicalize` process, which is normally mapped to the object of an active clause. But, because the clause is embedded and is an argument of the `Require` relation, another constraint specifies that `Input` must be the subject of the clause (because of the control relation). In order to satisfy these two constraints, the passive voice is selected for the embedded clause, to make of the `affected` role the subject.

SURGE is capable of performing such complex syntactic inference mainly because unification is bidirectional and works well as a constraint propagation mechanism.

[45]Example (2) was provided by Karen Kukich.

5.1.10. Summary

In summary, a syntactic realization grammar provides abstraction from multiple grammatical forms. It encapsulates knowledge of syntactic constraints. As a consequence, the input to the grammar does not need to specify a detailed syntactic structure, ordering constraints, morphological features and agreements, closed-class words such as prepositions, copulas, and determiners.

The input to SURGE is a semantic form, whose structure is justified by linguistic generalizations. This level of abstraction makes the task of the lexical chooser much easier. In the next sections, I proceed to a description of the specific interface provided by SURGE to the lexical chooser.

5.2. Syntax of the Clause

The clause grammar in SURGE is mainly an implementation of systemic linguistics, specifically [Halliday 85] and [Fawcett 87], but, as lexical choice issues took more importance in the ADVISOR II system, my view of syntax has also been influenced by more lexicalist linguistic theories, in particular HPSG [Pollard & Sag 87] and Meaning-text Theory as presented in [Mel'cuk and Pertsov 87] and used in [Polguere 90]. I start this brief review by presenting the systemic model of the clause I have implemented in SURGE, and then integrate this model with a more lexicalist perspective.

A clause is a configuration of roles around a central process realized in language. Process is used here in the systemic sense and has a very general interpretation - independent of the contrast event/process. Processes include relations, states, events and actions. For example, the possessive process type is characterized by the combination of two participants: possessor and possessed. The syntax of the clause determines how a given process type is to be realized by a linguistic structure by specifying in which order constituents appear, the category of each constituent (NP, verb group, PP or adverbial), and imposing certain realization features on governed constituents (prepositions of PPs, mood of embedded clauses).

There are tree types of decisions involved when processing a clause, which are encoded in three systems:

- **Transitivity system**: determines which configuration of participants is acceptable, and imposes a syntactic category on the realization of each participant.
- **Voice system**: maps the participants to grammatical functions like subject or object and imposes an order on the constituents. The voice system controls the choice of passive vs. active, but also choices like the dative shift, dislocations and clefts which affect the position of constituents in the clause by purely syntactic means.
- **Mood system**: determines if the clause is interrogative, imperative, assertive or non-finite (infinitive or participial).

Of these three systems, the transitivity system is the one most closely linked to the lexical chooser; the other two systems are located in the syntactic realization module. So I concentrate mainly on the transitivity system in this chapter.

5.2.1. The Transitivity System and Composite Processes

At the basis of the transitivity system is a distinction between three types of complements in the clause:

- **Participants**: are most integrated in the clause. They are realized by NPs without prepositions in grammatical functions subject and object and are inherent participants of the process.
- **Circumstantials**: are less integrated in the clause. Circumstantials specify the location of the process, its manner, purpose etc. They are most often realized by PPs or adverbial adjuncts.
- **External Functions**: appear outside of the clause structure but correspond to one of the participants or circumstantials in the underlying semantic process. External functions are used in dislocations and clefts:

- Left Dislocation: *AI, I think you should definitely take.*
- Cleft: *It is AI that you should take.*

The *AI* constituent is external to the surface clause but corresponds to a participant of the process *you should take AI*.

External functions are closely associated with the pragmatic need to present information as new or in focus, and constructs like clefts and dislocations are selected by the voice system. So external functions are easy to recognize.

It is often less clear how to distinguish between circumstantials and participants. Three syntactic criteria are used to establish the participant-status of a role:

- **Un-movable**: a participant cannot be moved in the clause, circumstantials can be moved.

  You should take AI
  ** Should take AI you.*

  You should take AI this semester.
  This semester, you should take AI.

- **Obligatory**: a participant function must be filled in the clause for it to be complete. Circumstantials are optional.

  You should take AI.
  ** should take AI.*

- **Can become subject**: a participant can become the subject of an active or passive form of the clause. Circumstantials cannot.

  Maguire taught me the secrets of Operating Systems.
  The secrets of Operating Systems were taught to me by Maguire.
  I was taught the secrets of Operating Systems by Maguire.

  Maguire teaches Operating Systems in depth.
  ** In depth is taught Operating Systems by Maguire.*

The distinction, however, is not clearcut: These criteria are not all necessary to identify a participant. For example, in relational processes, *e.g., AI is difficult*, there is no notion of passive. So the third criterion does not apply to determine whether *difficult* is a participant or not. Similarly, a dislocation can authorize the movement of a participant, as in *Difficult, AI is.* And finally, certain participants can be present in a clause without being realized overtly as in *Take AI* (imperative form). But considering all three criteria together provides a good method to identify participants.

The transitivity system is, therefore, split at the top level between participants and circumstantials. Circumstantials can be added to any type of process in general, and, therefore, the choice and processing of circumstantials is independent of lexical choice. I therefore now concentrate on the part of the transitivity system processing participants.

A process type is defined as a configuration of participants. For example, the *material* process type is defined as a configuration of participants *agent* and *affected*. Semantically, a material process, therefore, corresponds to a predicate of the form *Process(agent, affected)*. Each participant role is further defined in semantic terms, by providing a ''definition'' for what it means to be the agent of a process for example. Such definitions are most clearly established in [Fawcett 87]. For example, the test for identifying an agent is to paraphrase the clause as in:

- *John took AI.*
- *What John did was to take AI.*

Only the experiential meaning of the original clause needs to be preserved by the paraphrase (of course, the discourse status of the participants is affected by the transformation, and so can be the interpersonal function of the clause).

The type hierarchy shown in Fig.5-5 shows the hierarchy of process types implemented in SURGE. Table 5-5 lists the participant roles defined for each type of process.

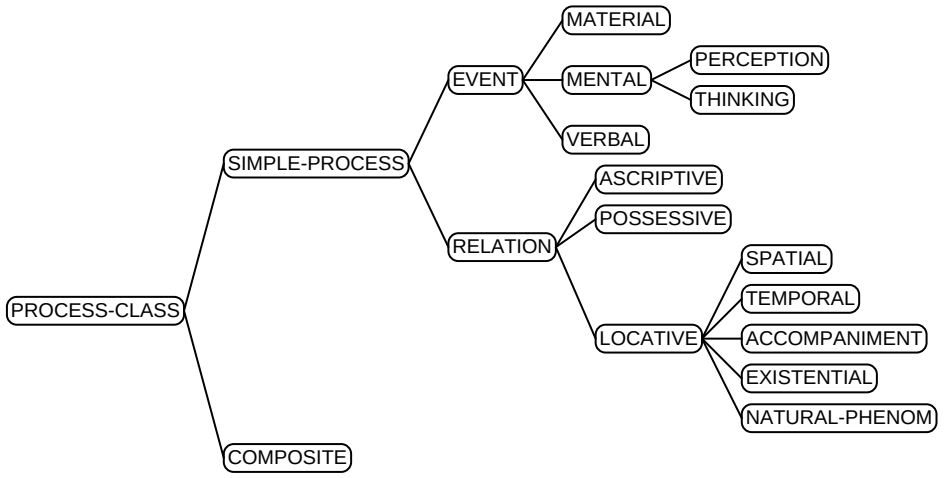


Figure 5-5: Process Types in SURGE

This table follows most closely the system proposed in [Fawcett 87]. It lists the different clause patterns corresponding to different process types. In addition to the process type, certain features indicate which subset of the participants must be present to allow the clause to be ''complete''. Thus, the material process type is characterized by three semantic features, which specify the perspective taken on an event in the world when describing it:

- **Agentive**: Yes or No. If the process is not agentive, the speaker presents the event as the result of some action, independently of the source that caused it to happen. When (agentive no) is set, affected is the only participant of the clause.
- **Effective**: Yes or No. If the process is not effective, the speaker presents the event independently of the effect it has on the world. When (effective no) is set, agent is the only participant in the clause.
- **Effect-type**: Dispositive or Creative. The effect of a material action can be to either modify an existing object or to create/destroy an object. When effect-type is dispositive, the role *affected* is defined, and corresponds to the object being modified by the action; when effect-type is creative, the role *created* is defined.

The relation process type can be specialized in two modes: *attributive* and *equative*. The attributive mode is used when an object (called the *carrier* of the relation) is related to a quality (called the *attribute*), *e.g., the box is blue*. In the equative mode two objects are equated; one object is the *identified*, the other one the *identifier*, *e.g., a box is a container*. In general, the equative mode is reversible while the attributive is not.

Since process types form a type hierarchy, where, for example, *locative* is a specialization of *relation*, there is a corresponding hierarchy of participant roles: so the roles of a relation, carrier and attribute are specialized into *possessor* and *possessed* for the *possessive* relation, *located* and *location* for the *locative* relation.

The transitivity system presented so far corresponds roughly to the coverage of NIGEL. There are, however, more clause patterns which are not covered by these simple transitivity patterns and that I support in SURGE using Fawcett's model. Examples of such complex patterns are:

Logic makes AI difficult.
She made him a good husband.

Process Type	Participants	Example
Material	Agent + Affected	John eats a pie.
	Agent	John runs.
	Agent + Created	John cooks diner.
	Affected	The sun shines.
	Created	The window popped.
Mental	Processor, Phenomenon	I think that it's good.
	Processor	I think.
Verbal	Sayer	It talks.
	Sayer + Addressee	John talks to Steve.
	Sayer + Verbalization	Steve says: fix it.
	Sayer + Addressee + Verbalization	Steve asks Doree to fix it.
Ascriptive	Carrier	I am.
	Carrier + Attribute	AI is difficult.
	Identified + Identifier	Maguire is OS's teacher.
Possessive	Carrier/Possessor + Possessed	AI has 5 assignments.
	Identified/Possessor + Possessed	Steve owns a box.
Locative	∅	It rains.
	Carrier/Located	There is a unicorn.
	Carrier/Located + Location	Steve is in his office.
	Identifier/Located + Location	Steve's office contains a computer.

Figure 5-6: Structural patterns for Simple processes

I went home.

For such sentences, there is an ''extra'' argument which does not fit in any of the simple process types reviewed so far. Fawcett's idea is that such complex processes can be viewed as the composition of two simple processes - one material and one relational. So, for example, the previous examples can be rephrased as:

Logic causes AI to be difficult.
She caused him to be a good husband.
I caused myself to be home.

The transformation consists in making explicit the causation captured in the semantics of the complex clauses. The agent role is viewed as the ''causer'' of a relational state. When the two processes of cause and relation are merged, one participant involved in both processes is merged. So, for example, *Logic makes AI difficult* is interpreted as having a transitivity structure of *Agent + Affected/Carrier + Attribute*, where *AI* is both the *affected* role of the causal process and the *carrier* role of the attributive process. Table 5-6 lists the complex processes that are generated by combining the different material patterns (Agent, Agent+Affected, Agent+Created, Affected, Created) with the different relational patterns. Sixteen new composite patterns are made available in the transitivity system.

Composite processes allow one to view a single verb, like *make* in *Logic makes AI difficult*, as the realization of two semantic relations. They are, therefore, a significant step in recognizing the non-isomorphic relation between linguistic structure and conceptual structure.

Process Type	Participants	Example
Material + Ascriptive/attributive	Agent+Affected/Carrier+Attribute	They made him rich.
Material + Ascriptive/equative	Agent+Affected/Identified+Identifier	They made him the boss.
	Agent/Carrier+Affected+Attribute	She made him a good wife.
Material + Possessive	Agent+Affected/Carrier+Possessed	The boss gave the babe cold cash.
Material + Locative	Agent+Affected/Carrier+Location	I push the box to the left.
Material/Creative + Ascriptive	Agent+Created/Carrier+Attribute	He cooked the diner spicy.
Material/Creative + Locative	Agent+Created/Carrier+Location	The program popped the window on the screen.
Material + Ascriptive	Agent/Carrier+Attribute	He became rich.
Material + Ascriptive/Equative	Agent/Identified+Identifier	He became the boss.
Material + Possessive	Agent/Carrier+Possessed	The boss bought a Rolls
Material + Locative	Agent/Carrier+Location	He went home.
Material + Ascriptive	Affected/Carrier+Attribute	The kettle boiled dry.
Material + Possessive	Affected/Carrier+Possessed	The boss received cash.
Material + Locative	Affected/Carrier+Location	The box fell on the floor.
Material/Creative + Ascriptive	Created/Carrier+Attribute	He was born blind.
Material/Creative + Locative	Created/Carrier+Location	The window popped on the screen.

Figure 5-7: Structural patterns for Composite processes

5.2.2. Transitivity vs. Subcategorization

While systemic linguists have developed such complex transitivity systems to account for the structure of the clause, lexicalist approaches to grammar have developed the notion of subcategorization. The idea is that in any phrase, the lexical head determines the structure and order of the subconstituents - or dependents in a dependency-based grammar like MTT - of the phrase. To capture this dependency, the HPSG grammar describes each lexical item capable of governing arguments with a feature called the subcategorization list, called *subcat*. The subcat of a verb is a list of the constituents the verb expects to form a complete phrase. So for example, the verb *to deal* as in *AI deals with logic* is represented in the lexicon with a subcat *deal[Subcat(<NP> <PP:with>)]* which indicates that to produce a clause headed by this verb, one must find an NP and a PP headed by the preposition *with*. In HPSG, the subcat is also related to the semantics of the verb and the semantic function of its arguments as shown in the example in Fig.5-8.

Every linguistic constituent in HPSG is associated to a feature structure containing a feature *cont* describing the information contents realized by the linguistic element. In the figure, the verb realizes a conceptual relation *chase(chaser, chasee)*. The coindexing of the variables *x* and *y* indicates that the content of the syntactic complement filling the first slot of the subcat of the verb is *chaser*, and the second slot is filled by the realization of *chasee*.

Thus the subcategorization approach provides roughly the same type of information as the transitivity system in systemic grammars: mapping from semantic roles to syntactic arguments, determination of the syntactic category of each argument and of the order of the arguments. The differences are that the subcat feature describes a single lexical entry whereas the transitivity approach aims to describe general classes of semantically similar verbs and that

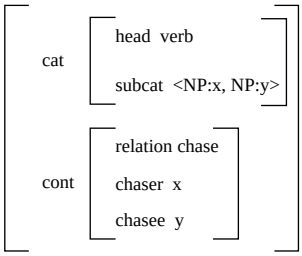


Figure 5-8: Subcategorization in HPSG (from [Sag and Pollard 91, p.74])

the HPSG approach associates this information with a lexical item whereas the systemic approach associates it with a semantic input.

While developing the SURGE grammar, I have found that both approaches are convenient. In a sense, the transitivity system offers a generalization over sets of verbs, by defining classes of semantic processes and corresponding classes of syntactic behavior. So SURGE accepts both inputs described in terms of transitivity processes and inputs providing their own subcat feature, with an explicit mapping between the roles and the syntactic constituents. To allow for this possibility, a new process type, called *lexical* has been added to the process-type hierarchy, as shown in Fig.5-9.

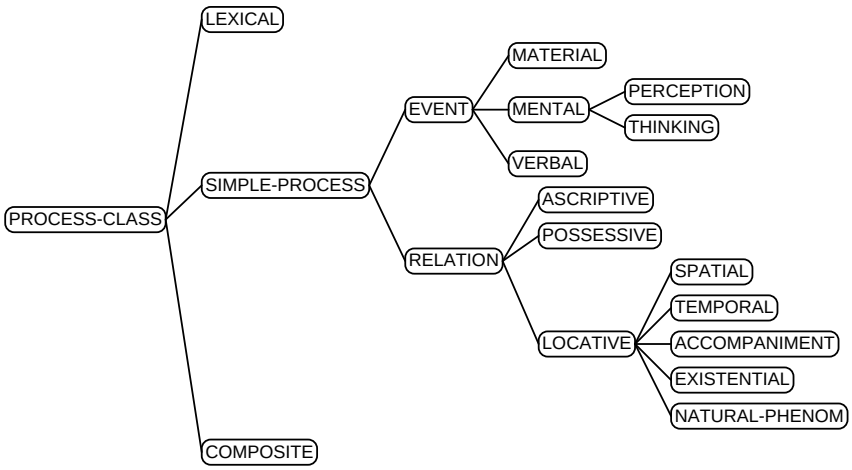


Figure 5-9: Process Types in SURGE with the Lexical branch

The top level part of the grammar distinguishes between lexical, simple and composite processes, as shown in Fig.5-10. Inputs corresponding to each type of specification are shown in Fig.5-11.

In the grammar, when the process is of type lexical, the oblique frame of the clause is unified with the subcat feature of the head verb. The oblique frame is a notion derived from HPSG. In the oblique frame, elements are ordered from 1 to 3 for inherent participants, and adjuncts come afterwards, as elements 4, 5 etc. The obliqueness order does not correspond to surface order, but rather to a measure of closeness to the heart of the process. So PP complements are more oblique than NPs, and objects are more oblique than subjects. The oblique frame is more general than the labels of syntactic functions like subject, object, indirect-object etc. In contrast, when the process is of type simple or composite, the grammar performs the mapping from the participants to the slots of the oblique frame.

```
(def-alt transitivity
  (:index {process type})
  (:demo "Is the process lexical, simple or composite?")
  (((process ((type lexical)))
    ;; Need to have the mapping
    ;; lexical-roles -> oblique in lexicon
    ;; The mapping is under the subcat feature of the process.
    (process ((subcat given)))
    (oblique {^ process subcat}))

   ;; Following are the general classes of verbs
   ;; using Halliday's and Fawcett's transitivity system.
   ((process ((type simple-process)))
    (:! simple-process))

   ((process ((type composite)))
    (:! composite-process))))
```

Figure 5-10: Top level alt in the transitivity region of SURGE

Thus, one can interpret a transitivity process as a generalization over a class of similar subcategorization frames. In fact, the role of the transitivity system can be seen as mapping a transitivity description into a specific subcategorization frame. When the input specifies a process of type lexical, the transitivity system is simply shortcut.

The differences between processes specified by simple transitivity, complex transitivity and lexical subcategorization are illustrated in Fig.5-11. In the first example in Fig.5-11, the process is of type simple possessive. It is characterized by a configuration of roles {possessor, possessed}. In this case, the grammar does most of the work ''automatically'': selection of the default verb ''to have'' to express the possession in attributive mode (the default is ''to own'' in equative mode) and mapping of possessor to subject and possessed to object.

The second example, *Logic makes AI difficult*, is of type composite: it is the merging of a material action *Logic does something to AI* and of an ascriptive relation *AI is difficult*. This is captured by describing *Logic* as the agent,[46] *AI* as both the carrier and the affected, and *difficult* as the attribute. In this case again, the grammar does most of the work on its own: selection of the default verb ''to make'' to express the process material/ascriptive and assignment of the semantic roles to the slots 1, 2 and 3 of the oblique frame of the clause.

Finally, the third example, *John wants to learn AI*, is of type lexical. In this case, the arguments of the process do not appear under the feature participants, which is restricted to general semantic labels of well-known transitivity classes, but instead under the feature lex-roles. In addition, the process contains a feature subcat, which indicates the mapping between lex-roles and the subcat slots 1 and 2. In this example, the lex-roles are *influence* and *soa* (for state-of-affair). In addition to establishing the mapping lex-roles->subcat, the subcat feature imposes syntactic constraints on the complements. Thus, the syntactic complement realizing the soa role is constrained to be a clause, of mood infinitive. More interestingly, the control behavior of the verb is also described in this feature. Note indeed that in the example, there is a case of control between the subject of the matrix clause (*John wants soa*) and the embedded clause (*John learns AI*). This description of control follows the model presented in [Sag and Pollard 91]: the oblique frame of the embedded clause is constrained to have a gapped subject (slot 1 of the feature oblique has a feature (gap yes)) and the subject is constrained to be coindexed with the subject of the matrix clause (with the equation unifying {process subcat 2 oblique 1 index} and {subcat 1 index}). This equation, therefore, captures the control relation between the two arguments. This control behavior depends on the verb *to want* and such a complex constraint is difficult to implement in general terms in the grammar, as demonstrated in [Sag and Pollard 91]. So

[46]Note that Logic is not animate but is still described as an agent. There is some controversy in systemic linguistics as to whether agents must be animate. In any case, as discussed in the next chapter, I do not take this argument structure description as a description of conceptual relations, but more as a deep-syntactic representation of the clause. In the next chapter, I propose a derivation of this clause from the conceptual relations used in the ontology that does not rely on the notion of agent as an animate entity.

```
;; A process of simple type possessive
(def-test simple
  "AI has six assignments."
  ((cat clause)
   (process ((type possessive)))
   (participants ((possessor ((cat proper) (lex "AI")))
                  (possessed ((cat common)
                              (lex "assignment")
                              (cardinal ((value 6)))))))))


;; A process of type composite: material + ascriptive
;; Contains both participants of material and ascriptive type.
;; The verb is selected by the grammar by default.
(def-test composite
  "Logic makes AI difficult."
  ((cat clause)
   (process ((type composite)
             (relation-type ascriptive)))
   (participants ((carrier ((cat proper) (lex "AI")))
                  (attribute ((cat adj) (lex "difficult")))
                  (agent ((cat proper) (lex "Logic")))
                  (affected {^ carrier})))))


;; A process of type lexical
;; The subcat explicitly maps the semantic roles to syntactic
;; arguments in order of obliqueness and pre-selects syntactic
;; features of the complements.
(def-test w1
  "John wants to learn AI."
  ((cat clause)
   (process ((lex "want")
             (type lexical)
             (subcat ((1 {^3 lex-roles influence})
                      (2 {^3 lex-roles soa})
                      (2 ((cat clause)
                          (mood infinitive)
                          (oblique ((1 ((index {^4 1 index})
                                        (gap yes))))))))))))
   (lex-roles
     ((influence ((cat proper) (lex "John")))
      (soa ((proc ((type mental) (lex "learn")))
            (participants
             ((phenomenon ((cat proper) (lex "AI"))))))))))
```

Figure 5-11: Example of inputs for simple, composite and lexical process types

encoding it in the lexical entry for the verb is a good strategy. Another reason to prefer a lexical entry over a transitivity entry for a given verb is when the verb does not fit neatly into the semantic categories offered by the current transitivity system. For example, the verb *to require* would be best described as a mental process. But when used in a context such as *AI requires many assignments*, it is problematic to view *AI* as the processor of a ‘‘mental process’’. In contrast, viewing this verb as a lexical process does not require any such forced interpretation.

So in summary, the realization clause grammar of SURGE can take as input either a general semantic description relying on well-known transitivity relations, or a lexical description headed by the verb, relying on the subcategorization of the verb. Other aspects of the syntax of the clause are determined by other features of the input. I briefly discuss these features in the next subsection.

5.2.3. Other Aspects of the Syntax of the Clause

There are two other aspects of the syntax of the clause besides the transitivity system which are of importance: the mood system, which determines whether the clause is assertive, interrogative, imperative or relative and the modality to be used in the verb group; and the voice system, which can move constituents around to satisfy focus constraints, by using devices such as cleft, dislocation and passivation. In systemic terms, the mood system is related to the interpersonal meta-function, which means that the choices are constrained by features that relate to the function of the clause as a message between persons, whereas the voice system is related to the textual meta-function, that is, the features it uses relate to the function of the clause as a part of an extended discourse.

The mood system of SURGE deals with the different moods shown in Fig.5-12. At the top level, the choice is between finite and non-finite, *i.e.*, between a clause with a conjugated verb and a clause with a verb in a non-conjugated form. Examples of clauses at each mood are shown in Fig.5-12.

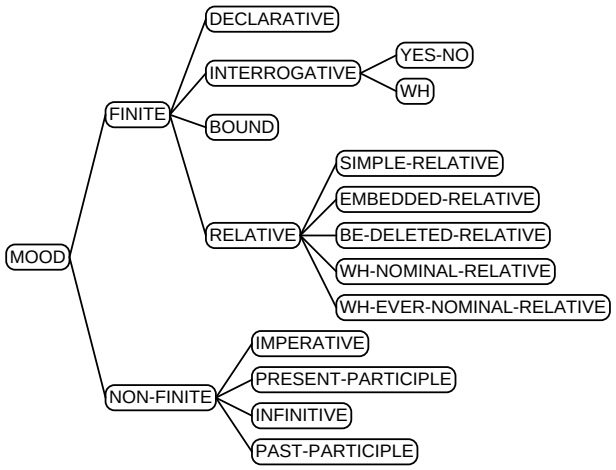


Figure 5-12: Moods in SURGE

For wh-interrogative and relative clauses, a constituent of the clause is displaced from the regular position, and replaced by either an interrogative or relative pronoun. The constituent which is the scope of this transformation is identified by being conflated with the *scope* feature. Examples on input illustrating the scope feature are shown in Fig.5-14.

Note that in the case when the scope feature is used, it is conflated with a semantic role, and not with the syntactic constituent realizing it. Thus, the grammar must maintain the realization links between syntactic constituents and corresponding semantic constituents to recognize the scope in the surface structure of the clause and replace it with a pronoun. This is implemented using a feature called *realization*. The realization link is always a pointer from a semantic constituent to the syntactic constituent realizing it. This is a general mechanism which is also used in the lexical chooser, as discussed below.

So in summary, the main input required to control the mood system is the feature mood and the feature scope when required.

For the modality, SURGE distinguishes between deontic and epistemic modality, using the classification shown in Fig.5-15, derived from [Palmer 86]. The single feature modality determines which modality is to be used. Examples of modality usages are shown in Fig.5-16.

Mood	Example
declarative	*AI has six assignments.*
interrogative yes-no	*Does AI have six assignments?*
interrogative wh	*Who teaches AI?*
bound	*whether AI has six assignments.*
simple-relative	the person *who teaches AI.*
embedded-relative	the topics *with which AI deals.*
be-deleted relative	the topics *covered in AI.*
wh-nominal relative	I know *who teaches AI.*
wh-ever-nominal relative	*whoever teaches AI.*
imperative	*Take AI.*
present-participle	*the person taking AI.*
infinitive	*for him to take AI.*
past-participle	*the topics taught in AI.*

Figure 5-13: The different moods used in SURGE

Finally, the voice system in SURGE implements passivation, but not dislocation and clefts. The input feature controlling the voice decision is the feature *focus* which is conflated to one of the semantic constituents of the input. A focus conflation forces its target to become the subject of the clause. An example is shown in Fig.5-17.

5.2.4. Features expected by the Syntactic Realization Grammar for a Clause

To summarize, the following features in the input to the syntactic realization grammar of SURGE are needed to generate a clause:

- A process of type either simple, composite or lexical. If the process is lexical, its lexical head must be specified. Otherwise, for simple and composite processes, default verbs are selected by the grammar. These defaults can be overriden if desired.
- A configuration of roles corresponding to the type of the process.
- The mood feature and if necessary the scope feature pointing at one of the roles.
- The modality feature.
- The focus feature pointing at one of the roles.

Appendix A lists example inputs exercising the coverage of the clause region of the SURGE syntactic realization grammar.

```
(def-test wh
  "Who teaches AI?"
  ((cat clause)
   (mood wh)
   (process ((type material) (lex "teach")))
   (scope {^ participants possessor})
   (partic ((agent ((animate yes)))
            (affected ((cat proper) (lex "AI")))))))


;; Cover interpreted as a locative equative [abstract location]
(def-test relative
  "The topics which AI covers."
  ((cat common)
   (head ((lex "topic")))
   (number plural)
   (definite yes)
   (qualifier
     ((cat clause)
      (scope {^ participants located})
      (process ((type locative) (lex "cover") (mode equative)))
      (participants ((location ((cat proper) (lex "AI")))))))))


(def-test long-distance
  "The person who you think won the award."
  ((cat common)
   (head ((lex "person")))
   (animate yes)
   (qualifier
    ((cat clause)
     (proc ((type mental) (lex "think")))
     (scope {^ participants phenomenon participants possessor})
     (participants
      ((processor ((cat personal-pronoun) (animate yes)
                   (person second) (number singular)))
       ;; Win interpreted as "get": Agent/Carrier+Possessed
       (phenomenon
        ((cat clause)
         (process ((type composite)
                   (relation-type possessive)
                   (tense past)
                   (lex "win")))
         (participants
          ((agent {^ possessor})
           (possessed ((cat common) (lex "award")))))))))))))
```

Figure 5-14: Input FDs with a displaced constituent

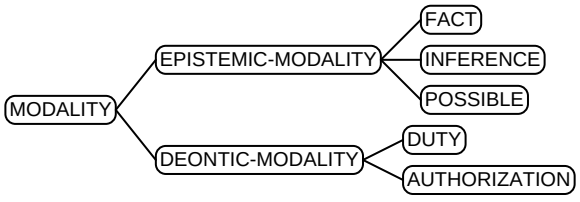


Figure 5-15: Modality in SURGE

```
;; modality possible: generate can
(def-test modality1
  "When properly done, it can be safe."
  ((cat clause)
   (process ((type ascriptive)))
   (modality ((value possible) (type epistemic)))
   (partic ((carrier ((cat personal-pronoun)
                      (gender neuter)
                      (person third)
                      (index id1) ; unique identifier for this referent
                      (number singular)))
            (attribute ((lex "safe") (cat adj)))))
   (circum ((temporal-background
             ((cat clause)
              (position front)
              (mood past-participle)
              (process ((type material) (lex "do")))
              (partic ((affected ((index {^5 partic carrier index})))))
              (circum ((manner ((lex "properly")))))))))))

;; modality duty: generate must
(def-test modality2
  "You must do it."
  ((cat clause)
   (modality ((value duty) (type deontic)))
   (process ((lex "do") (type material)))
   (partic ((agent ((cat personal-pronoun)
                    (animate yes)
                    (person second)
                    (number singular)))
            (affected ((cat personal-pronoun)
                       (gender neuter)
                       (person third)
                       (number singular)))))))

;; modality inference: generate must
(def-test modality3
  "It must be interesting."
  ((cat clause)
   (modality ((value inference)))
   (proc ((type ascriptive) (mode attributive)))
   (partic ((carrier ((cat personal-pronoun)
                      (gender neuter)
                      (person third)
                      (number singular)))
            (attribute ((lex "interesting")))))))
```

Figure 5-16: Effect of using modality

5.3. Syntax of the NP

The syntax of the NP is much less elaborate in English than the syntax of the clause, and it has received much less attention in generation work.[47] The systemic grammar of [Halliday 85] presents the following top-level structure for the noun group:

```
Deictic Numerative Epithet          Classifier Thing  Qualifier
                   Attitude Quality
those   two        splendid old     electric   trains with pantographs
```

[47]For example, the Nigel grammar documentation does not mention any treatment of the noun group.

```
;; Focus is put on the affected role which becomes subject.
(def-test focus
  "AI is made difficult by Logic."
  ((cat clause)
   (focus {^ participants affected})
   (process ((type composite)
             (relation-type ascriptive)))
   (participants ((carrier ((cat proper) (lex "AI")))
                  (attribute ((cat adj) (lex "difficult")))
                  (agent ((cat proper) (lex "Logic")))
                  (affected {^ carrier})))))
```

Figure 5-17: Effect of focus selection on voice selection

In SURGE, I have adopted a similar description provided in [Winograd 83, Appendix B.]:

```
determiner-sequence describers classifiers head qualifiers
```

The head of a noun group is a noun. The types of syntactic categories for the head as used in SURGE are shown in Fig.5-18. As for other constituents, the head determines which types of modifiers can be present in the group. For example, proper nouns and pronouns do not accept describers in general and do not require a determiner. But all the categories shown in the figure can fill the same positions within container clauses - so they are all considered as instances of the same category. The SURGE grammar enforces the appropriate co-occurence restrictions between types of modifiers and category of the head.

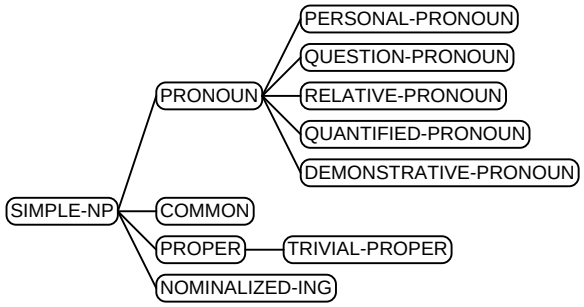


Figure 5-18: Types of noun phrases in SURGE

As for modifiers, I give a detailed description of the determiner sequence structure in the next section. The other elements of the noun group are the pre-modifiers and the post-modifiers. Two sorts of pre-modifiers are distinguished: describers and classifiers. Describers can be used as the complement of an ascriptive clause like:

- *an interesting class: the class is interesting*
- *a difficult assignment: the assignment is difficult.*

In contrast, classifiers cannot be moved in the same way:

- *an AI assignment: *the assignment is AI.*
- *a programming class: *the class is programming.*

When several describers are present, their ordering before the head obeys complex rules, for which no completely satisfying account has been found. Apparently, the adjectives denoting the more ‘‘inherent’’ properties are placed closer to the head, but this inherentness may be defined in context (cf. [Teyssier 68] for the basic argument for the

correlation between classifying and determinative function of adjectives and closeness to the head, [Quirk *et al* 72, pp.924-927] for the general inherentness argument, [Vendler 68, Chap.VIII] for a classification of adjectives and its use in ordering of describers, [Ney 83] for a critique of Vendler's approach and an alternative analysis based on a statistical analysis and proposing habit strength as the major determinant of adjective ordering). Since the determination of the order of describers relies on conceptual information, the syntactic realization component of SURGE does not impose any constraints on ordering and expects an ordered list of describers in its input. It is, therefore, the task of the lexical chooser to order the describers appropriately.

When several classifiers are present, the structure of the modification is also difficult to predict. [Winograd 83, p.515] gives the example of *a toy factory* (a factory producing toys) vs. *a toy* factory] (a toy which has the shape of a factory). I get back to this issue of noun-noun modification in Sect.6.5.4.1 when I discuss which conceptual elements are mapped to such linguistic structures. SURGE also does not impose any constraint on order of classifiers and expects an ordered list of classifiers in input.

Two sorts of post-modifiers are distinguished: restrictive and non-restrictive. Restrictive modifiers take part in the identification of the denotation of the noun phrase, while non-restrictive ones add information about an already identified denotation. For example, *the AI topics that are interesting* denotes a subset of the set of AI topics (restrictive modification), while *the AI topics, which are interesting* denotes the whole set of AI topics and informs the hearer that all these topics are interesting. The feature `(restrictive yes)` in the input to SURGE indicates which type of qualifier must be realized.

Describers are generally adjectives, present or past participles. A short phrase can also be used [Winograd 83, p.515]:

a *larger than usual* crowd
a *soon to be released* book
a *man eating* tiger

When participles are used, one can distinguish between objective and subjective modification:

an *interesting* class: the class interests X
an *interested* student: X interests the student

In general, a present participle describer is a subjective modifier (the head is the subject of the predicate) while a past-participle describer is an objective modifier (the head is the object of the predicate).

Qualifiers can be relative clauses, participle clauses, preposition phrases or noun groups in apposition. Restrictive relative clauses are introduced either by the pronoun *that* or by no pronoun at all, as in, *the class you took last semester*. Non-restrictive relative clauses are introduced by a wh-relative pronoun, and are generally preceded by a comma. PP qualifiers are restrictive. Appositions are non-restrictive.

Vendler [Vendler 68, p.85-87] argues that describer adjectives are also to be interpreted as restrictive modifiers, and that the pattern *A N* can be interpreted as *N that is A*. There is, however, a class of describers where this equivalence does not hold. For example:

a nuclear scientist
* *a scientist that is nuclear*
a scientist that works on nuclear topics

a beautiful dancer
* *a dancer that is beautiful*
a dancer that dances beautifully

Such modifiers are called non-predicative modifiers and are very close in nature to noun-noun modifiers. I also get back to the structure of these modifiers in Sect.6.5.4.1 to determine which conceptual relation they realize.

Finally, there is a whole class of noun phrases best seen as nominalizations of clauses, *e.g., parental refusal, the flight of the plane, the destruction of the city by the barbarians*. Such constructs are studied in depth in [Vendler 68, pp.24-82] and in [Levi 78, pp.167-221]. In such constructs each of the modifiers of the nominal group can be

mapped to a transitivity role of the underlying clause, *e.g., city* is the affected, *barbarians* is the agent. The constructions of such NPs from a clause description would be a valuable addition to a generation grammar, but it is currently not implemented in SURGE.

The structure of the NP is thus characterized by the determiner, describer list, classifier list, head and qualifier list. In addition the following features characterize the agreement behavior of the NP and correspond to semantic distinctions that are syntactically or morphologically marked:

- **Definite**: yes or no.
- **Animate**: yes or no.
- **Number**: singular, dual or plural.
- **Gender**: masculine, feminine or neuter.
- **Case**: subjective, objective or genitive.
- **Person**: first, second or third.
- **Countable**: yes or no.

More features of this type which have to do with the determiner sequence are listed in the next section.

In summary, the structural input expected by the syntactic realization of SURGE to produce an NP consists of the following features:

- **Cat**: one of the categories shown in Fig.5-18.[48]
- **Head**: a simple element of category noun or pronoun.
- **Determiner**: a complex constituent described in the next section.
- **Describer**: an ordered list of pre-modifiers of category adjective or participle. Participle modifiers can be subjective or objective.
- **Classifier**: an ordered list of pre-modifiers of category noun or adjective. Classifiers are non-predicative modifiers.
- **Qualifiers**: an ordered list of post-modifiers of category clause (which is mapped to an appropriate form of relative clause), prepositional phrase or appositive noun phrases.

An example input for an NP is shown in Fig.5-19.

```
(def-test np1
  "The interesting OS class which Maguire teaches."
  ((cat common)
   (head ((lex "class")))
   (definite yes)
   (describer ((cat verb) (modifier-type subjective) (lex "interest")))
   (classifier ((lex "OS")))
   (qualifier ((cat clause)
               (process ((type lexical) (lex "teach")))
               (scope {^ lex-roles 2})
               (lex-roles ((1 ((cat proper) (lex "Maguire")))))))))
```

Figure 5-19: Example of syntactic input for an NP

Note that in SURGE, most of the features of the NP constituent are inherited from the head subconstituent. So, for example, the input FD `((cat common) (lex "box"))` is equivalent to `((cat common) (head ((lex "box"))))`, with the `lex` feature inherited from the head. Note also that all the features have reasonable defaults and only need to be specified when these defaults must be overriden.

[48]Note that SURGE does not deal with pronominalization. Pronouns must be indicated in the input.

5.4. Syntax of the Determiner Sequence

The determiner sequence is in itself a complex constituent. It has the specificity that it is mainly a closed system: that is, the lexical elements are part of a small set of words which are determined completely by semantic and syntactic features. The issue when devising a grammar is to determine a minimal set of features accounting for the variety of determiner sequences observable in English. I first review the approach proposed in [Halliday 85, pp.159-176] which proposes a simple manageable set of semantic features controlling the realization of the determiner sequence, but leaves out some of the observable syntactic complexity. I then discuss the description of the syntax of the determiner sequence presented in [Quirk *et al* 72, pp.136-165] which covers more of the syntactic variety. The result is the identification of a set of features that control the realization of the determiner sequence in SURGE.

5.4.1. Halliday's Analysis

Halliday presents the structure of the determiner sequence as:

```
deictic    deictic2  numerative
determiner adjective numeral
the        same      three      topics
```

The function of the deictic element is to identify whether a subset of the thing is denoted, and if yes, which subset. The subset can be identified by different means: deixis and distance (near or far from the speaker - *this* vs. *that*), possession (*my*, *John's*) or not at all (*the*). Figure 5-20 shows the system of features used by Halliday to distinguish between the different forms of deictic element used in the determiner sequence.

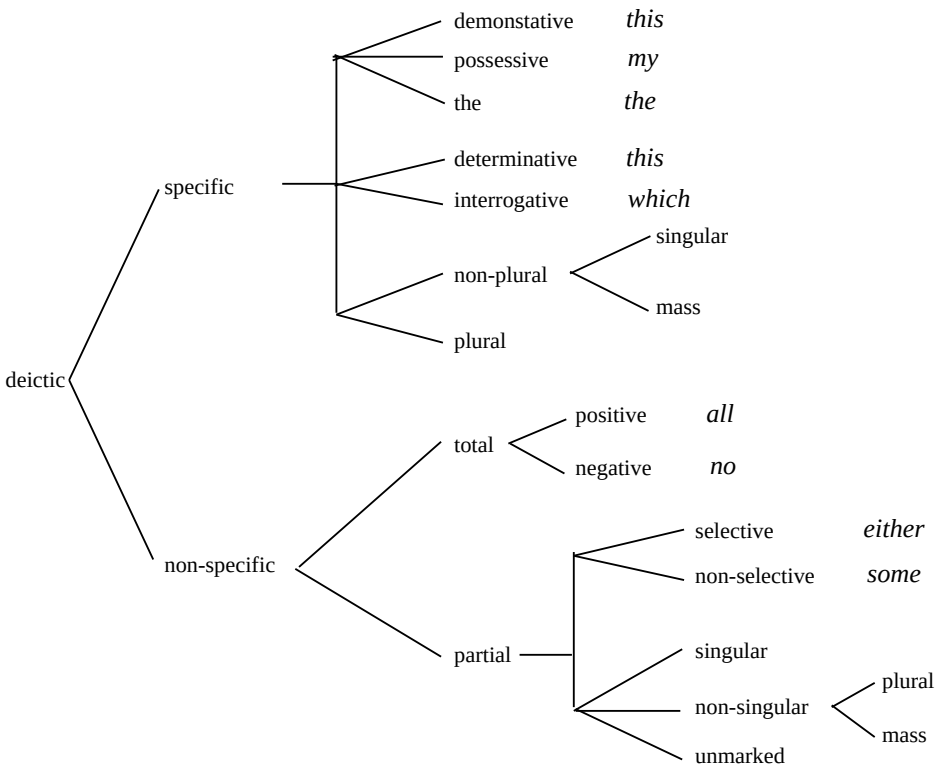


Figure 5-20: System of features controlling the deictic element in the determiner sequence

The top level distinction is between specific and non-specific determination. A specific deictic denotes a known, well-identified subset of the thing denoted by the head of the noun group. A non-specific deictic denotes a subset identified by quantity. Specific deictics can be either determinative or interrogative and plural or non-plural.

Non-specific deictics are either total or partial. Total deictics denote either the whole set denoted by the thing, or the empty set. They are *all* or *no*, and *both*, *each*, *every*, *neither*. Partial deictics denote a non-empty strict subset of the set denoted by the thing. They come in two sorts: selective and non-selective. Selective deictics have a sort of partitive meaning: *one*, *either*, *some* (as in *some people*) or *any*. Non-selective deictics are the indefinite *a* and *some* as in *some cheese*.

Halliday distinguishes between the number system in the two different contexts where it is applicable: specific and non-specific|partial, by noting that the contrast is between plural and non-plural in specific deictics and singular, non-singular and unmarked for partial deictics. Unmarked number deictics are *no*, *some* (as in *some people*) and *any*. This distinction is important when working on the mapping from conceptual to syntactic input, because it highlights the difference between conceptual number (cardinality of a set) and linguistic number, which corresponds to a speaker's decision to express the quantity.

The function of the deictic2 element is to specify the subset of the thing by ''referring to its fame, familiarity, its status in the text or its similarity/dissimilarity to some other designated subset'' [Halliday 85, p.162]. Examples of uses of the deictic2 function are: *the same class*, *the usual three assignments*. The deictic2 is an adjective but it is viewed as part of the determiner sequence and not of the describer list because it can occur before the numerative part, in contrast to any other describer, which must occur after the numerative part.

Finally, the numerative part system is shown in Fig.5-21. The function of the numerative element is to express the quantity or the place of thing. The numerative can be either quantitative (expressing a quantity, *e.g.*, *three*) or ordinative (expressing a relative position, *e.g.*, *third*). In both cases, the expression can be either exact or inexact. Exact numeratives correspond to numerical values (1, 2, 3...). The source of the inexactness can be an approximation device (*about three*, *roughly third*, *approximately ten*) or a range expression (*between six and ten*). Alternatively, it can be a context dependent expression like *the next*, *many*, *few*, *more*. There are many such inexact quantifiers and Halliday does not provide a more detailed classification of this class of words.

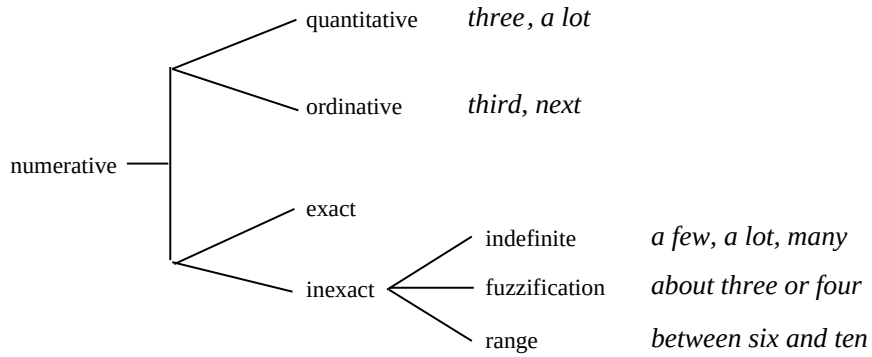


Figure 5-21: Numerative system in the determiner sequence

Halliday's system provides a very useful set of features to control the realization of the determiner sequence, but it does not discuss co-occurence restrictions between the different elements of the sequence (*e.g.*, *a lot* is a complete sequence that fills only the numerative slot, *many* can be used alone or in combination with a deictic, etc); and it does not provide a complete coverage of the syntax of determiners (*e.g.*, account for sequences like *all of the first ten students*) and a finer classification of the inexact quantifiers. So I looked at other syntactic descriptions to complete this first analysis.

5.4.2. Quirk *et al*'s Analysis

The grammar in [Quirk *et al* 72, pp.136-165] provides a detailed analysis of the determiner sequence, reproduced in a simpler form in [Winograd 83, B.2.2.1]. The prototypical structure proposed is:

pre-determiner	(of)	determiner	ordinal	cardinal	quantifier	
all	of	the	first	ten		*commandments*
half	of	my			many	*parents*
twice as					much	*work*

The first discrepancies with Halliday's system are the presence of the pre-determiner, the absence of the deictic2 element and the co-occurence of ordinal and cardinal elements in certain configurations. SURGE authorizes the presence of all these elements together.

Pre-determiners can be one of the following elements: *all, both* or *half*, multipliers (*twice, three times*) or fractions (*one fourth*). Multipliers occur with count and mass nouns and with singular count nouns denoting a number or an amount (*double their salaries, twice his strength*). Fractions can be used with or without an of-construction. In SURGE, I have enforced the use of an *''of''* after a fraction. Fractions can co-occur with the predeterminer *all* for mass nouns.

The ordinal element follows the determiner and comes before the quantifier. There are two forms, corresponding to Halliday's exact/inexact contrast: *first, next, last, another* can cooccur with a cardinal element or the quantifier *few* and a plural countable noun; *second, third* etc. cannot co-occur with a quantifier or cardinal and require a singular countable noun.

The cardinal and quantifier element are mutually exclusive. They both express the quantity of the noun group thing, and the difference between them corresponds to the exact/inexact contrast of Halliday. There is a closed-system set of quantifiers summarized in Fig.5-22 and an open-system set of quantifiers of expressions of the form *a good deal of, an incredible amount of* etc. The open-system quantifiers exhibit the same structure as partitive constructions of the form *a cup of tea*. Such constructions are particularly interesting because the head of the noun group does not correspond to the head of the conceptual element they realize, but to the element expressing the quantity. In addition, when they are modified, there is a form of transfer, where the adjective syntactically depends on the head, but semantically modifies the thing, as in: *a strong cup of tea* vs. *a cup of strong tea*. These constructs have been analyzed for text generation in [Dale 88, Sect.5.5.3]. I have not implemented generalized partitives in SURGE, but instead focused on these open-class quantifiers which have the same surface structure.

The determiner slot can be filled by elements of category article, demonstrative, question determiner, possessive determiner or quantifier. This corresponds to Halliday's classification with features demonstrative, possessive, determinative and interrogative, except that the quantifier determiner class is not explicitly covered by Halliday's system as a determiner. Winograd's classification of quantifier determiners is shown in Fig.5-22, which lists the different classes with the corresponding set of features in Halliday's system when available. This table shows that Halliday's system does not cover the evaluative determiners and does not distinguish between degree and existential quantifiers. In SURGE, I have added two features to distinguish between those elements: degree and evaluative.

Finally, [Quirk *et al* 72, pp.156-159] lists special cases of noun classes that take zero articles, including seasons, institutions, transport means, illnesses. These classes of head nouns are recorded in SURGE, and the feature head-denotation-class in the FD describing the thing is used to indicate if the thing falls into one of these categories.

There is an important distinction between specific and generic reference which I have not implemented in SURGE.

5.4.3. SURGE's Grammar for Determiners

I have integrated some of the observations found in [Quirk *et al* 72] with Halliday's system when implementing SURGE's grammar for the determiner sequence. The set of features controlling the realization of the determiner sequence in SURGE is:

	positive	comparative	superlative
plural countable	*a / the / my,... / whose / these/those* { *great / good* } *many*	*more*	*most*
	few	*fewer*	*fewest*
	a few	——	——
	several	——	——
mass	*the / my,... / whose / this/that* *little*	*less*	*least*
	a little	*more*	*most*

Figure 5-22: Closed system quantifiers
From [Quirk *et al* 72, p.144]

Type	Example	Halliday's features
Universal	*all, every, both, each*	total positive
Negative	*no*	total negative
Existential	*some, any, several*	partial
Degree	*many, much, most, few*	?
Evaluative	*enough, too many, too much*	?

Figure 5-23: Classification of quantifier determiners

- **Definite**: yes, no.
- **Countable**: yes, no.
- **Partitive**: yes, no.
- **Interrogative**: yes, no.
- **Possessive**: yes, no.
- **Number**: singular, dual, plural.[49]
- **Reference-number**: singular, dual, plural.
- **Distance**: far, near.
- **Selective**: yes, no.
- **Total**: +, -, no.
- **Exact**: yes, no.
- **Orientation**: +, -, none.

[49]Dual is necessary to account for the determiners *both*, *either* and *neither*.

- **Degree**: +, -, none.
- **Evaluative**: yes, no.
- **Evaluation**: +, -.
- **Status**: none, same, different, mentioned, specific, usual, entire or a lexicalized item.
- **Case**: objective, subjective, genitive.
- **Head-cat**: pronoun, common, proper.
- **denotation-class**: quantity, season, institution, transportation, meal, illness.
- **Comparative**: yes, no.
- **Superlative**: yes, no.
- **Cardinal**: number.
- **Ordinal**: number or last, next (+), previous (-).
- **Fraction**: numerator, denominator.

The following features have not appeared in the previous discussion of the syntax of the determiner sequence: reference-number, orientation and degree. Reference-number corresponds to the cardinality of the set to which a selective makes reference. Recall that the function of the deictic element is to specify a subset of a larger set. I call this larger set the reference set, and reference-number describes the cardinality of this set when it must be expressed by the determiner. For example, the NP *neither class* or *either class* can only be used when the reference-number is dual (*either* is a selection of one element out of two).

The length of the list of features necessary to control the realization of the determiner sequence is due to the fact that the determiner sequence is mainly a closed system, where all lexical differences must be determined by some configuration of features. The grammar dealing with the determiner sequence is one of the most complex part of the SURGE, with 109 decision points (approximately a fifth of all decision points in the grammar). Note that not all features need to be present at the same time in the input, and that most features receive a default value when not specified. Appendix B shows a list of example inputs exercising the coverage of the grammar for determiners in SURGE. All these features can appear at the NP level and are sent to the determiner constituent as needed by the grammar.

5.5. Summary and Conclusions

This chapter has presented the fragment of SURGE which the lexical chooser of ADVISOR II can account for. SURGE is a large portable syntactic realization grammar, which has been used in many generation projects both at Columbia and at many outside research labs.

In the ADVISOR II system, the input of SURGE must be constructed by the lexical chooser. Therefore, this chapter serves as an introduction to the next chapter on lexical choice by specifying what the output of the lexical chooser must be.

In this chapter, I have first outlined the function of a syntactic realization grammar, listing nine tasks that SURGE performs:

- Define syntactic structure.
- Provide ordering constraints among the syntactic constituents of the sentence.
- Propagate agreement features between the words of the sentence, providing input to the morphology module to compute inflections and conjugations.
- Select closed-class words.
- Prevent over-generation.

- Provide ways to express possible inputs for as many syntactic forms as possible (coverage).
- Provide defaults for syntactic features.
- Control grammatical paraphrasing
- Perform syntactic inference.

These nine tasks combine to define an abstract perspective on syntax. The overall goal of this chapter was precisely to describe the abstract view of syntax that SURGE presents to the lexical chooser, by specifying which features are required to drive SURGE, at each syntactic rank.

Three syntactic ranks were reviewed in this chapter: clause, NP and determiner. The fragment of the clause grammar described comprises three main systems: the transitivity system maps a semantic process description and a configuration of semantic roles onto a linguistic subcategorization frame; the voice system maps the subcategorization frame to syntactic functions such as subject or object; the mood system determines if the clause is interrogative or assertive etc. The main innovations in the clause grammar are:

- A complete coverage of *composite* transitivity processes, accounting for sentences such as *John painted the box blue*.
- The integration of lexical subcategorization with a systemic transitivity system.

Both of these features allow the lexical chooser to gather two conceptual relations into a single clause. In addition, the clause grammar covers complex syntactic phenomena such as long-distance dependencies and control.

At the NP level, SURGE requires the specification of a head possibly modified by three types of complements: classifiers, describers and qualifiers. In addition, a determiner sequence is derived by SURGE from the quantification and reference features of the NP.

Finally, the determiner level is most remarkable because it is mostly a closed-system: it generates words from purely syntactic features, without requiring any open-class items in the input. The main contribution of the SURGE grammar for determiner is its large coverage: 24 features are used to control the generation of a large variety of determiner sequences, including inexact determiners such as *many, a large amount of* etc.

The SURGE grammar as a whole is notable for its robustness and wide coverage. Its main function is to integrate existing grammatical theories into a highly usable, portable and consistent system. Extensive experience with SURGE, by many researchers and in many different domains, has demonstrated that this goal has been largely achieved.

SURGE is also the largest generation grammar developed based on the Functional Unification Grammar formalism. The development of such a large scale grammar has demonstrated the validity of the formalism for the expression of syntactic knowledge. It has also demonstrated the efficiency of the FUF implementation and the usefulness of the extensions I have presented in Chap.4.

This mainly descriptive chapter has defined the level of semantic and syntactic description that the grammar can interpret. It therefore serves as a basis for the next chapter on lexical choice, which explains how conceptual information extracted from a knowledge base and pragmatic information such as argumentative evaluations can be mapped onto the level of description presented here.

Chapter 6
Linguistic Decisions and Lexical Choice

This chapter presents the lexical choice method that has been implemented in ADVISOR II. The chapter focuses on the decisions that are involved when expressing an argumentative evaluation. Argumentative evaluations are an instance of floating constraints: they can be realized by linguistic constituents at many ranks. In this work, connectives, verbs, adjectives and determiners are considered. Therefore, the lexical choice procedure presented focuses on the issue of cross-ranking realization; that is, how the same input semantic element is mapped onto different linguistic ranks, a problem not previously addressed in generation.

Lexical choice is the part of the realization process that determines which words appear in the output. Since lexical items come with constraints on which syntactic structures are compatible with them, lexical choice also implies a choice of syntactic structure. So I distinguish between two aspects of lexical choice:

- The lexical chooser selects heads out of the conceptual network, and for each head, determines its syntactic category and which syntactic constituents depend on the head.
- The lexical chooser selects an entry from the lexicon and maps each head to an actual word or sequence of words.

In linguistic terms, these two dimensions correspond to *syntagmatic* and *paradigmatic* selections respectively: the first dimension defines how a semantic structure is mapped onto a linguistic structure, and dependency relations are created among linguistic constituents; the second dimension selects one out of a set of substitutable items for inclusion in the linguistic structure.

An important aspect of the conceptual to linguistic structure mapping, which was mentioned in Sect.4.1.3 (p.88 *ff*) when discussing floating constraints, is that the two structures are non-isomorphic: the same semantic input constituent can be mapped at several levels in the linguistic structure (*i.e.*, the head of a conceptual relation is not necessarily the head of a linguistic constituent), and several semantic elements can be mapped onto the same linguistic constituent. In terms of lexical choice, this mismatch implies that the same semantic input can be realized by lexical material at different ranks - clause, noun-group or even some aspects of the meaning of a single word, a phenomenon that I call *cross-ranking realization*.

In order to adequately handle cross-ranking realization, the lexical chooser must be able to take a wide variety of structural decisions within the clause. I call the task of constructing a clause structure from a conceptual network *clause planning*. Similarly, the construction of an NP structure from a set description or from a conceptual network is called *NP planning*. The term *planning* is used here to highlight the analogy with *text planning* performed at the paragraph or discourse level. The main issue in both text and clause or NP planning is one of *information packaging*: determining which conceptual relation or entity is to become the head of the linguistic realization, and how the other relations are attached to the head (using rhetorical relations at the paragraph level and syntactic relations at the sub-sentence level); determining which information gets included; determining the size of each chunk of information to be included in each linguistic constituent; these decisions being all performed under pragmatic constraints.

I focus in this work on a specific class of floating constraints: the argumentative evaluation of a variable in the underlying logical form. Because the structural attachment of argumentative evaluations is largely under-constrained, they provide a good testbed of the cross-ranking capabilities of a lexical chooser. For example, in the following sentences, the same argumentative evaluation is realized at different syntactic ranks:

(1) AI requires many assignments.

(2) AI has many assignments, so it could be difficult.
(3) AI requires 6 assignments, which is a lot.
(4) AI is interesting but it has many assignments.

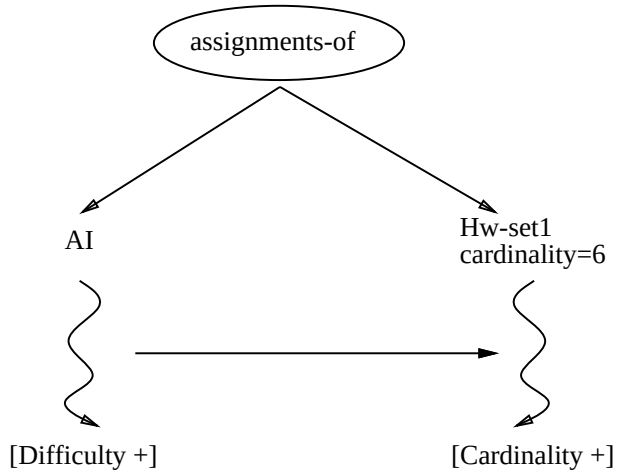


Figure 6-1: Conceptual input with floating argumentative constraints

The input conceptual form and argumentative evaluation which can be realized by these four sentences is shown in Fig.6-1. It consists of a conceptual network expressing the relation between the AI class and a set of assignments, annotated by an argumentative evaluation specifying that AI is difficult because of the number of assignments (a composite evaluation). Argumentative evaluations are depicted using wavy lines. In (1), the evaluation of AI as difficult is realized by selecting the verb *require* which carries as a connotation that its subject is difficult; the evaluation of the number of assignments is realized by selecting the judgment determiner *many*. In (2), the AI evaluation is realized by selecting the predicative adjective *difficult*. In (3), the number evaluation is realized by adding a relative clause to the NP *6 assignments*. In (4), the evaluation of AI as difficult is realized by selecting the connective *but*, and connecting it to the evaluation of AI as interesting. Since *but* requires a contrastive relation to hold between the left and right clauses, the second one is interpreted as an evaluation of AI as difficult.[50]

In these examples, the same argumentative evaluations in the input are realized at different syntactic ranks: main verb, predicative adjective, relative clause, connective and determiner. This floating effect is the main object of this study. To understand how this variation can be produced, one must first determine how each site in the syntactic tree can become a potential realization site for an argumentative evaluation. Then, one must determine where to attach the argumentative evaluation among the potential sites.

Since this work focuses on cross-ranking realization, the task of the lexical chooser is defined here as the mapping of a purely conceptual network onto a linguistic structure. The input is not committed to any linguistic categorization, so that each input element can be mapped to different linguistic ranks. In this framework, many different mappings can be considered. Pragmatic constraints on lexical choice must be considered in order to select an appropriate linguistic form. Figure 6-2 illustrates the role of the lexical chooser and which pragmatic constraints are considered in this work.

In this chapter, I present the lexical chooser as follows: I first present an overview of the lexicalization task, highlighting the structural (syntagmatic) decisions that must be made when choosing lexical entries to cover a conceptual network. This analysis defines the steps of clause planning and NP planning. Section 6.2 describes the input to the generator, a semantic representation that is uncommitted to linguistic and lexical decisions, so that the same input can be realized using widely different linguistic resources, and, therefore, allows for cross-ranking

[50]This interpretation relies on the activation of three topoi: (1) *the more a class is difficult, the less a student wants to take it*, (2) *the more a class is interesting, the more a student wants to take it*, and (3) *the more a class has assignments, the more it is difficult*.

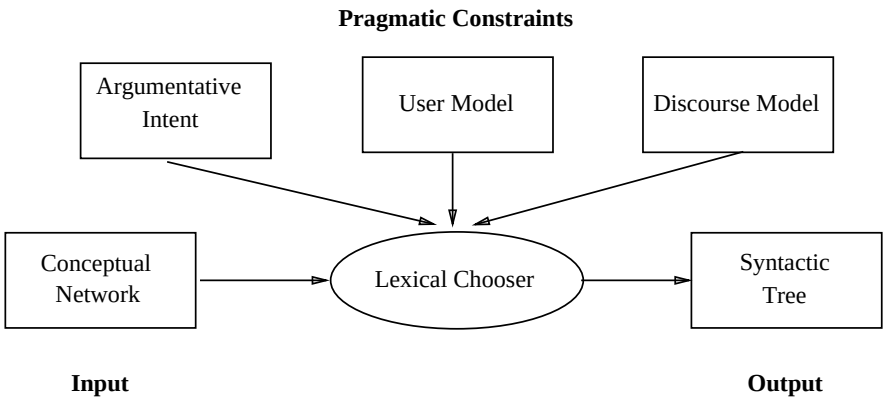


Figure 6-2: The role of the lexical chooser

realization. Section 6.3 presents the overall control strategy used to perform lexical choice, comparing it to generators developed previously. The following five sections describe the type of decisions made during the generation of clauses, nominal groups, determiner sequences, adjectives and connectives. In each of these sections, I follow the same pattern of presentation: the list of conceptual elements which can be realized at the given rank is first enumerated; then the list of syntactic features required by SURGE to generate a constituent at the given rank is presented. These two steps define the input and output of the lexical chooser. Finally, I explain how the lexical chooser maps from the conceptual input to the syntactic output, showing the form of the lexical entries used in each case. I specifically identify the places where argumentative evaluations in the input constrain the lexicalization. In conclusion (Sect.6.9), I summarize the constraints that have been identified on lexical choice.

In many cases, I present a list of alternative realizations for the same input. Whenever possible, I have associated these sets of candidates with pragmatic features controlling the choice of the ‘‘most appropriate’’ realization. In some cases, however, I have not found ways to select among the alternative realizations. In these cases, the choice is made arbitrarily, and this results in greater variety of output and greater paraphrasing power.

6.1. Lexical Choice and Cross-Ranking Realization

6.1.1. Structural Decisions in Lexical Choice: Sub-sentence Planning

I explain in this section which structural decisions are performed by the lexical chooser and how these decisions depend on available lexical resources. Structural decisions are taken at two levels in the linguistic structure: clause and noun group. I call these decisions respectively clause and NP planning. In this section, I first briefly describe the form of the conceptual input to the lexical chooser, and then give examples of structural decisions at each of these levels illustrating how pragmatic factors control their selection.

6.1.1.1. Input to the Lexical Chooser

First, I briefly present the type of input available to the lexical chooser. In the ADVISOR II domain, consider the following conceptual relations, retrieved by the expert system in response to the query: *Should I take AI this semester?*

1. AI has 6 homework assignments.
2. Among the AI assignments, there are 2 essays.
3. Among the AI assignments, there are 4 programming assignments.
4. AI covers logic.

5. Logic is a mathematical topic.

6. AI covers expert systems and knowledge representation.

Assume the following facts about the student are known to the system:

1. The student is interested in expert systems and knowledge representation.

2. The student does not have experience in writing essays.

3. The student does not have experience in mathematics.

4. The student is interested in programming.

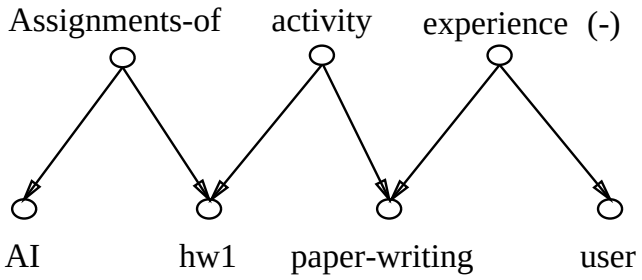


Figure 6-3: A conceptual network

All of these facts are actually encoded in a conceptual (*i.e.*, non-linguistic) knowledge representation formalism. The elements of this formalism are individuals (*e.g.*, the AI course), sets (*e.g.*, the set of AI assignments), and binary relations (*e.g.*, the topics relation between a course and a set of topics). A graphical representation of the conceptual relations listed above is shown in Fig.6-3. The task of the lexical chooser is to map such a network onto a lexicalized tree, where heads are identified by a lexical entry and a syntactic category (clause, noun group or adjective) and their daughter nodes are all instantiated. Such trees are later processed by a realization grammar that turns them into sentences. In this chapter, I will not show the lexical trees and instead show the corresponding final sentences.

6.1.1.2. Clause Planning

One aspect of lexicalization at the clause level is to determine a traversal of the input conceptual network. For example, different traversals of the network in Fig.6-3 yield the following sentences:[51]

- *You do not have experience in writing essays which two assignments of Intro to AI require.*
- *You have experience in programming which four assignments of Intro to AI require.*
- *Two topics which Intro to AI covers, expert systems and NLP, interest you.*
- *Intro to AI covers two topics, expert systems and NLP, which interest you.*

These four (ugly) sentences correspond to traversals of the conceptual network starting at different nodes and following different relations. These sentences illustrate an initial lexicalization strategy: relations are mapped to clauses, individuals and sets are mapped to noun groups. When a node is shared between two relations, a relative clause is generated so that two relations end up modifying the same noun group. Each relation is mapped to a verb, *e.g.*, the relation `topics` to *cover*, and each individual to a noun, *e.g.*, the concept `AI` to *Intro to AI*.

To improve on this first algorithm, I have developed a series of techniques, aiming at making the output more fluent by taking advantage of lexical resources in a more sophisticated manner. The first limitation I address is the

[51] All example sentences in this chapter, including the ugly ones (which are of course included for contrast only:-), have been generated by FUF.

assumption of a strict mapping between relation and clause, and entity and noun phrase. A second aspect is to allow for the generation of other types of NP modifiers, besides relative clauses. A third aspect is to allow for the expression of argumentative evaluations in the language being generated.

At the clause level, I define the notion of perspective: a choice of perspective in the conceptual network consists of selecting one relation as the head of the clause to be generated. For example, different perspective choices produce the following sentences:

- *You do not have experience in writing essays which two assignments of Intro to AI require.*
- *Two assignments of Intro to AI require writing essays, in which you do not have experience.*
- *Intro to AI requires two assignments which require writing essays, in which you do not have experience.*

These three sentences cover the same three conceptual relations, but the head relation is different in each one. Note that as a consequence of this choice, the same relation can be lexicalized by a verb when it is at the toplevel of the clause structure (*e.g., Intro to AI requires ...*) or as a prepositional phrase in other cases (*e.g., the assignments of Intro to AI*). The notion of perspective is related to the notion of focus as used, for example, in [McKeown 85], but still distinct: the perspective is a relation in the conceptual network while the focus is an entity. Once the perspective is chosen, the focus can be shifted between the participants of relation, by switching the order of the complements, as in the following sentences:

- *You do not have experience in writing essays which two assignments of Intro to AI require.*
- *Writing essays, which is required by two assignments of Intro to AI is unfamiliar to you.*

These two sentences adopt the same perspective on the conceptual network, by choosing the `experience` relation as the head node, but the order of the participants is changed: in one case a different lexicalization is used (*X has experience in Y* vs. *Y is (un)familiar to X*); in the other case, the transformation is performed by syntactic means, by choosing the passive voice (*A requires X* vs *X is required by A*). The decisions determining the structure of the toplevel clause in the sentence are described in more detail in Sect.6.4. They are captured by the definition of the realization features `perspective` and `focus` and depend on the set of entries in the lexicon that can describe conceptual relations as main verbs of clauses.

Allowing the perspective to be freely selected in the input conceptual network among the many relations it contains is key to cross-ranking realization. In contrast, most existing text generators require the input to specify which relation is the perspective, and will, therefore, be mapped to the toplevel clause, and which relations are ‘‘entity modifiers’’. This distinction imposes a linguistic structure onto the semantic representation even before any linguistic knowledge is accessed.

6.1.1.3. NP Planning

At the NP level, the way to improve the flexibility of the mapping from conceptual to linguistic structures is to allow for the construction of complex nominals as opposed to using relative clauses only. A complex nominal is a noun group modified by predicative and non-predicative modifiers, using syntactic tools such as nominalization, noun-noun modification and adjective-noun modification. The lexical chooser must be able to decide when such modifications are appropriate to cover a given conceptual relation. Examples of complex nominals are:

Without complex nominals: *Intro to AI requires four assignments which require programming.*
With complex nominals: *There are four programming assignments in AI.*

Note that when entities become modifiers of complex nominals, like *programming* in *programming assignments*, they are less susceptible to become the focus of subsequent clauses. So the decision to use complex nominals is related to the choice of perspective and of focus, and to topic progression in the paragraph. I use the notion of *recoverably deletable predicate* of [Levi 78] to determine when noun-noun modifiers can be used as explained in Sect.6.5.

The lexicalization of sets into noun groups includes structural decisions similar to the choice of perspective at the clause level. For example, consider the set of topics containing the three elements `expert-systems`, `knowledge-representation` and `vision` and defined in intension as the set of topics covered in Intro to AI such that the user is known to be interested in them. The following lexicalizations illustrate the variety of forms that can be generated, depending on which aspect of this definition needs to be highlighted:

Many interesting topics covered in AI - vision, expert systems and knowledge representation.
Vision, expert systems and knowledge representation.
Three AI topics.
Interesting topics.
Three interesting topics.
Three AI topics that you should find interesting.

Section 6.5.1 discusses the lexicalization of sets, enumerates 54 different patterns of lexicalizations for the semantic representation of sets I have defined, and defines a set of four realization features which control which form is generated among the 54 syntactic possibilities.

The features `focus` and `perspective` at the clause level and `realize-quantity` at the NP level are examples of *realization features*, *i.e.*, features controlling the structure of the linguistic constituent constructed from the conceptual network. These features are set by the lexical chooser by considering various pragmatic constraints. In this work, I have focused on the influence of argumentative intent on lexical choice, which I now discuss.

6.1.2. Realizing Argumentative Evaluations

The consideration of perspective and focus and the possibility to produce complex nominals significantly improved on the naive lexicalization algorithm presented at the beginning of this section, by using more sophisticated syntactic modification than just clause/complement and noun-group/relative clause. The resulting linguistic structure is much richer than any work proposed so far in generation in that it allows cross-ranking realization: a relation in the conceptual network can be mapped to a clause, a nominalized noun group, a non-predicative modifier of a complex nominal an adjective pre-modifier or prepositional post-modifier of a noun. A set of realization features have been identified which constrain which structural mappings can be used.

I now introduce argumentative evaluations to constrain the value of these realization features, and thus indirectly lexicalization of the conceptual network. Chapter 2 defined the notion of argumentative evaluation with its relation to topoi. Argumentative evaluations can be considered as annotations added to the input conceptual network which are added to express the argumentative intent of the speaker. Argumentative evaluations, therefore, play a double role in the lexicalization process: they constrain how the underlying conceptual network can be lexicalized and they must themselves be lexicalized.

To illustrate how argumentative evaluations constrain lexical choice for the affected semantic network, consider, for example, the conceptual relation expressing the fact that the AI course has 6 assignments. Different argumentative evaluations can be applied to this set of assignments. Recall from Chap.2 that I distinguish between three modes of evaluation: absolute, relative and composite, as illustrated by the following sentences:

- Absolute: *AI has 6 assignments.*
- Relative: *AI has more assignments.*
- Composite: *AI has many assignments, so it is difficult.*

To convey composite and relative evaluations, the lexical chooser forces the determiner to express the cardinality qualitatively (using *more* or *many*).

This decision can in turn change the structure of the syntactic constituent bearing the evaluation. Consider, for example, how the choice of different orientations on the same scale can induce different structural mappings:

- *Only one AI topic, logic, is about mathematics, so AI should not be theoretical.*
- *AI covers logic, which is very mathematical, so it could be quite theoretical.*

When the evaluation of the cardinality of the set of topics is positive (*i.e.*, the cardinal is viewed as a ‘‘large’’ number), the actual number, one, cannot be expressed, because all positive determiners require a plural noun phrase (*e.g.*, *many*, *a lot* etc). The decision not to express the quantity is expressed by setting `(realize-quantity no)` in the corresponding noun-group. In contrast, when the evaluation is negative, *i.e.*, one is viewed as a ‘‘small’’ number, the quantity is expressed, setting `(realize-quantity cardinal)`. When the quantity must be expressed, realizing the extension alone is not appropriate because it would produce a conjunction with no

determiner. Instead, an apposition must be generated, with a generic term serving only to carry the determiner followed by a conjunction of proper nouns: *one AI topic, logic*. Similarly, the pattern *X is about mathematics* is used when the evaluation `<mathematical ->` is active, but the pattern *X is mathematical* is used instead when the evaluation is `<mathematical +>`, relying on the lexical properties of *mathematical*, which is a degree adjective realizing the positive evaluation when it is used.

Consider now how the argumentative evaluation itself is conveyed, by choosing linguistic devices at different levels:

- *AI has many assignments, so it is difficult.*
- *AI has six assignments, which is a lot, so it is difficult.*
- *AI requires many assignments.*

The composite evaluation `<the more assignments in a class, the more difficult the class>` is applied to the conceptual relation stating that AI has 6 assignments. The composite evaluation is made of two sides - one evaluation on the cardinality scale of the set of assignments on the left and one evaluation of the AI individual on the difficulty scale on the right. The left evaluation can be realized by choosing a quantifying determiner (*many*) or by adding a relative clause to the noun-group realizing the set description; the right evaluation can be realized by attaching a connected clause with the appropriate connective or by choosing a verb with the appropriate connotation (*require*). The realization feature `realize-argumentation` determines at which level the argumentative evaluation is mapped in the linguistic structure.

6.1.3. Other Constraints on Lexical Choice

The range of realization features found in this work demonstrates the variety of constraints affecting lexical choice: the input logical form is the first constraint; argumentative intent is a second class of constraints that I consider in detail. While building the lexicon, I also encountered the need to consider other pragmatic factors which combine to determine the value of realization features:

- **Topic progression** in the discourse and communicative structure determine the perspective and focus on the conceptual network, and therefore influences the selection of roots in the conceptual network to serve as top level lexical predicates.
- **Knowledge of the user** determines whether certain parts of the logical form can be left implicit.
- **Politeness phenomena** constrain, for example, the selection of modals in the verb group and also influence the decision to include certain topics.
- **Style** criteria constrain the conciseness of the lexicalization, the depth of syntactic embedding that remains readable, the length of embedded constituents relative to the rest of the clause (balance criterion).

I have focused in this work on determining argumentative constraints on lexical choice, showing how argumentative features are derived from the content determination module and influence realization all the way to lexicalization and syntactic realization. I have determined the influence of these other pragmatic factors on lexicalization, without working on their derivation in the content determination module.

I have shown in this section that the key to more fluent generation is to make use of varied lexical resources. Specifically, an important requirement of the lexicalization process is that it be able to realize the same input constraint using linguistic devices at different ranks - a feature I have called cross-ranking realization. I now proceed to describe the form of the input, highlighting the fact that it is not committed to *a priori* linguistic devices and present specific lexical decisions implemented in ADVISOR II.

6.2. The Input

A key requirement to be able to perform cross-ranking realization is that the conceptual input to the lexical chooser must not be committed to any *a priori* linguistic categories. In ADVISOR II, the input to the lexical chooser is produced by the evaluation module. It consists of a network of conceptual relations extracted from the knowledge base enriched by argumentative evaluations. The size of the conceptual network is such that all the information supporting the argumentative evaluation is represented. A simple example of input is shown in Fig.6-4.

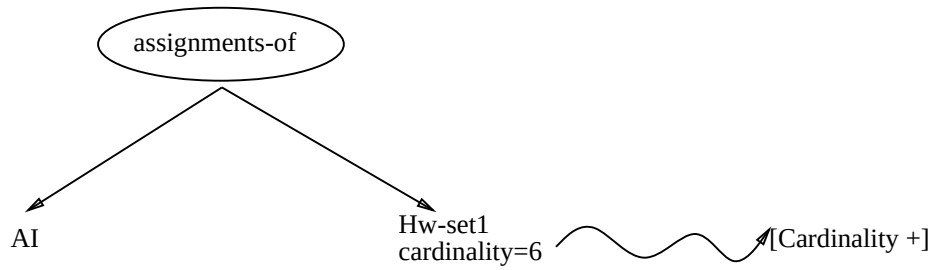


Figure 6-4: A simple input to the lexical chooser

This figure shows the conceptual relation of `assignments-of` holding between the AI course and a set of assignments whose cardinality is known to be six. This observation serves as a basis for the argumentative evaluation that AI has many assignments. The argumentative evaluation is shown as a wavy line in the figure. In the implementation, this input is represented in the FD formalism, as shown in Fig.6-5.

```
((cat topos)
 (justification
  ((rule eval-10)
   (assignments ((cat set)
                 (index hw1)
                 (kind ((cat assignment)))
                 (cardinality 6)))
   (class ((cat class) (index ai1) (name ai-class)))
   (class-relation ((cat relation)
                    (name assignments-of)
                    (1 {justification class})
                    (2 {justification assignments}))))))
 (left ((cat evaluation)
        (evaluated {justification assignments})
        (scale ((name "cardinality")))
        (orientation +)
        (type absolute)
        (value ((measure 6) (unit count))))))
```

Figure 6-5: FD representation of a simple input

I first focus on the description of the conceptual network, which appears under the attribute `justification`. The information is directly derived from a knowledge-base encoded in CLASSIC, a KL-ONE type knowledge representation system. A CLASSIC knowledge-base is made up of concepts and attributes, which are instantiated into instances and binary relations. For example, `ai-class` is an instance of the `class` concept. The `class` concept is defined by a set of attributes. One of these attributes is the slot `assignments-of` which links an object of type `class` to several objects of type `assignment`. An attribute is, therefore, a binary relation between the set of classes and the set of assignments. The conceptual network contains objects of three types:

- **Individuals** correspond to instances in CLASSIC. They are represented by an FD of the form:

```
((name <name-of-instance>)
 (cat <name-of-concept>)
 (index <unique-id>))
```

The index is used to uniquely identify each instance in the network. Each concept in CLASSIC is represented as a category in the FD. A FUF type hierarchy is derived to correspond to the inheritance relations between CLASSIC concepts, so that, for example, `(cat person)` can be unified with `(cat teacher)` if the CLASSIC knowledge base defines `teacher` as a specialization of `person`.

- **Sets** encode information about sets of instances in CLASSIC. They are represented by an FD with the following features:

```
((cat set)
 (index <unique-id>)
 (kind <representation-of-prototype>)
 (cardinality <n>)
 (extension <list-of-individuals>)
 (intension <a relation>)
 (reference <a set>))
```

Sets are also uniquely identified by an index. The `kind` feature is used when the set is a homogeneous collection of objects of the same type. The `cardinality` feature specifies the number of elements in the set when it is known, and the `extension` feature is a list of the individuals that are elements of the set. The features `intension` and `reference` are described in more detail in Sect.6.5.

- **Relations** correspond to attributes in CLASSIC. They are represented by an FD with the following features:

```
((cat relation)
 (name <name-of-attribute>)
 (1 <argument1>)
 (2 <argument2>))
```

By convention, argument 1 is the individual that ''contains'' the attribute, and argument 2 is the individual that is the value of the attribute in CLASSIC. The arguments of a relation can be sets. In this case, the semantics is distributive:
$S1RS2 \equiv \forall x \in S1, \forall y \in S2, xRy$.
I have not addressed the problem of representing more complex quantification schemes than the distributive one as discussed, for example, in [van Benthem 89] and [Keenan and Stavi 86]. Note also that in the ADVISOR II domain, I focus exclusively on sets of discrete elements as opposed to sets of ''substance'' characterized by packaging and quantity as was considered in [Dale 88, Sect.3.3]. A hierarchy of relations is also derived from the concept hierarchy encoded in the CLASSIC knowledge-base. A class of relations corresponds to each concept, *e.g.*, `class-relation` is the generic relation whose descendants are the attribute relations of the concept `class`, *e.g.*, `assignments`, `topics` etc.

In general, I distinguish between *relations* on one hand, and *entities*, which can be the arguments of a relation. Entities can be either individuals or sets. In this representation, objects do not have *properties*. Instead, there is a relation linking an object to an other object.

From the point of view of lexical choice, three characteristics of this representation make it different from the linguistic structure to be built:

- **Granularity:** the size of the conceptual network is not defined in terms of which linguistic constituent it encodes, but in conceptual terms only. That is, in most generation system, there is an assumption that the input to the lexical chooser is a chunk of conceptual material that is to be realized by a single clause. In such cases, the granularity of the input is determined by linguistic criteria. In ADVISOR II, the criteria are that the conceptual network must contain the full justification for an argumentative evaluation. It is, therefore, not linguistically motivated, and the input can be realized by a single clause or by several sentences. This has two advantages: first, it relieves the content planner from the burden of determining whether a network is too big to fit in a clause; second, it allows the generator to take more decisions in terms of clause planning.

- **Ontology:** the ontological primitives of the knowledge representation are not linguistically motivated. This is in contrast, for example, to the NIGEL approach which requires the ontology of the input to be a specialization of a linguistically motivated *upper-model* (cf. [Bateman et al. 90] and the discussion below in Sect.6.3). Instead, the ontology of the domain is motivated by the type of inferences that can be easily encoded.

- **Structure:** the structure of the input is not related to the syntactic category of the constituents that realize it. For example, entities can be realized by clauses or by noun groups, and relations, which encode basic semantic predication, can be realized by a clause, an adjective or a noun-noun modification.

These characteristics mean that the conceptual network is not committed to any linguistic decision and, therefore, can support cross-ranking realization and allows for more paraphrasing power.

The second facet of the input is made up of argumentative evaluations. The example in Fig.6-5 contains a single evaluation under the attribute `left`. I consider three types of evaluations, as discussed in Chap.2:

- **Absolute:** a measurable attribute is given an objective measure. For example, *AI has 6 assignments.*
- **Relative:** an attribute is measured relatively to a comparison class. For example, *AI has more assignments than Analysis of Algorithms.*
- **Composite:** an attribute is measured in terms of how it warrants an argumentative inference, using a topos. For example, *AI has a number of assignments such that it makes AI difficult* relies on the topos *the more a class has assignments, the more difficult it is.*

All three types of evaluations share the following features in their description:

```
((cat evaluation)
 (type <absolute | relative | composite>)
 (evaluated <a path to an entity in the conceptual network>)
 (scale <name of a scale>)
 (orientation <+ | ->)
 (value <v>))
```

The `evaluated` feature is a path to one entity in the conceptual network, which is the scope of the evaluation. This entity is evaluated on the scale identified by the feature `scale`. Scales identify an aspect or attribute of the evaluated entity. The scale `cardinality` denote the number of elements in a set. Otherwise, when the evaluated entity is a set, the semantics of the evaluation is again distributive:

$$evaluation(S,scale,+) \equiv \forall x \in S, evaluation(x,scale,+)$$

In ADVISOR II, primitive scales are identified by name. As discussed in Chap.2, there are seven primitive scales defined in the ADVISOR domain in addition to the domain-independent scale of cardinality.

The difference between the three types of evaluation resides in the description of the `value` feature, and in the interpretation of the `orientation` feature:

- **For absolute evaluations**, the value can simply be a measure, represented as follows:

```
(value ((measure 6)
        (unit count)))
```

  The unit reflects the method used to measure the value of the evaluated attribute. The unit `count` is used to count the elements in a discrete set.
  A feature `(orientation +)` gives to the evaluation the meaning of ''at least'', while `(orientation -)` means ''exactly''. In general, absolute evaluations when rendered in language exhibit a systematic ambiguity between the two interpretations (cf. for example [Klein 80, p.27], [Hirschberg 85, 5.1.4]). The ''at least'' interpretation of the evaluation is standard for cardinality because of the way set descriptions are constructed from the instances present in the CLASSIC knowledge-base: a relation `R(x,S)` where `x` is an individual and `S` is a set is produced when either a number restriction is provided in the concept hierarchy and can determine the number of fillers for the slot `R` in the instance `x` or more commonly when several instances of the relation $R(x, y_i)$ can be found in the knowledge base. In this second case, the set is known to have at least the elements found in the current state of the knowledge-base, but there is no guaranty of exhaustivity (which would correspond to a closed-world assumption in the knowledge representation). This corresponds to an ''at least'' interpretation of the evaluation of the number of elements in the set.

- **For relative evaluations**, a feature `comparison-class` is necessary to represent the elements to which the evaluated object is compared. The value of `comparison-class` is a set description. The `orientation` feature determines whether the comparison is ''more'' (orientation +) or ''less''

(orientation -). The value feature in this case is a path to the contrasted element in the set description. For example, in the representation of the relative evaluation realized by *AI has more assignments than Analysis of Algorithm*, the comparison class describes the single-element set *{Analysis-of-algorithm}*, defined as *{X s.t. X has a set of assignments A with |A|>|AI-assignments|}*. The notion of comparison class is described in more detail in Sect.6.7.

- **For composite evaluations**, `value` is a path to another evaluation. Recall that a composite evaluation has the form:

  ```
  <scale1(evaluated1), + <scale2(evaluated2), v>>
  ```

  That is, the value of the evaluation on `scale1` is the evaluation on `scale2`, or in other words, the value on `scale1` is such that the value on `scale2` is `v`. A composite evaluation must correspond to the instantiation of an existing topos of the form `/the more X is scale1, the more Y is scale2/`. The overall composite evaluation, therefore, has the following structure:

  ```
  ((cat topos)
   (left ((cat evaluation)
          (type composite)
          (evaluated <path1>)
          (scale <scale1>)
          (orientation +)
          (value {right})))
   (right ((cat evaluation)
           (type derived)
           (evaluated <path2>)
           (scale <scale2>)
           (orientation +)
           (value +|-)))
   (justification <conceptual-network>))
  ```

  The whole constituent is of category `topos`, because it must be compatible with the topoi-base activated in the current context. The features `left` and `right` correspond to the evaluations on the left and right-hand side of the topos. Justification is the conceptual observation which warrants the evaluation.

In summary, the input to lexicalization is, therefore, best viewed as a conceptual network, whose representation is not linguistically motivated, and which is annotated by argumentative evaluations.

6.3. Lexical Choice Strategy

This section discusses when lexical choice is performed in the overall realization process and how it interacts with the other realization decisions. I first review previous generation systems discussing different ways of positioning lexical choice in generation. I then present the approach followed in this work which consists of performing all lexicalizations that can be performed independently of syntax in a first stage, and keeping certain lexical decisions delayed until the syntax realization module is activated.

6.3.1. When is Lexical Choice Performed in Previous Work

The place of lexical choice within a complete generation system is a controversial topic. Several options have been put forth in previous work:

During Surface Realization, Interleaved with Syntactic Realization: In this approach lexical choice is considered as one linguistic decision like any other one, and, therefore, it is not distinguished as a separate lexical choice module. The grammar instead may contain very specific rules for lexical insertion. This approach makes most sense if the grammar used is part of the lexicalist family, viewing the grammar as a generalization or extension of the lexicon, for example HPSG [Pollard & Sag 87] or LFG [Kaplan & Bresnan 82]. As mentioned in Sect.4.1.1 (p.84), this is also the approach used in the semantic-head driven algorithm to generation [Shieber et al 90] and in most reversible approaches to generation. This approach is also adopted by systems using a phrasal lexicon, such as ANA [Kukich 83a], PHRED [Jacobs 85] and PAULINE [Hovy 88a]. When a phrasal entry is selected, all the syntactic

constraints between the elements of the template are already pre-selected, so there is no need for much syntactic realization (constituents are already ordered, their syntactic category is fixed in the template, closed class words are already selected, choices like passivation or dative-move are not considered). In general, in this approach, the semantic structure is traversed and a semantic head is selected and lexicalized by finding a corresponding lexicalization rule; after the head is selected, all the syntactic decisions that depend on this head are performed; then the dependent constituents are identified and lexicalized in turn. Lexicalization and syntactic realization are, therefore, interleaved as the linguistic structure is being built.

After Content Planning and Before Syntactic Realization: This approach is the most common among text generation systems. The surface realization component is separated into two successive modules: lexical choice followed by syntactic realization. The lexical choice module builds the linguistic structure and chooses all lexical entries. The syntactic realization component deals with agreements, ordering of constituents, choice of closed-class words, and in most systems can also perform syntactic transformations on the structure provided by the lexical chooser (like voice transformation, dative move, there-insertion or it-extraposition). This is the approach used in the TEXT system [McKeown 85], in MUMBLE [Meteer et al. 87], in COMET [McKeown et al 90], in SPOKESMAN [Meteer 89], in EPICURE [Dale 88] and in KALIPSOS [Nogier 90] for example.

It is also the basic strategy used in the GOSSIP system [Iordanskaja et al 91], which is based on Mel'cuk's Meaning-text theory [Mel'cuk and Polguere 87]. But since the underlying theory is highly stratificational, the process can be separated into three stages instead of just two:

1. Most of the structural decisions are made when converting the conceptual network (known as conceptual-communicative representations) into a SemR (semantic representation) first and then a DSyntR (deep syntactic representation) next. At this stage most open class items (nouns, verbs and adjectives) get selected.

2. Most closed-class items and grammatically determined words (*e.g.*, most prepositions and articles) are lexicalized when converting from the DSyntR to a SSyntR (Surface Syntactic Representation).

3. Open-class items which depend lexically but not syntactically on other lexical items are also selected at this stage (SSyntR construction) by using lexical functions (cf. [Mel'cuk and Polguere 87] for a discussion of lexical functions in the Meaning-text theory). For example, if an adverb is used to express the intensification of an action, the lexical function *magn* is inserted in the DSyntR instead of a specific lexical item. This lexical function is instantiated when building the SSyntR. Thus, depending on the verb that is eventually chosen, the appropriate modifier can be selected:

   - *Martin used Emacs a lot.*
   - *Martin made heavy use of Emacs.*

   In this example, the choice between *a lot* and *heavy* depends on the lexicalization of the predicate `use`. This dependence is captured by the use of a lexical function `magn(use)` in the DSyntR.

This separation has the benefit of clearly distinguishing between different types of lexical choice, and highlights the different types of dependencies that can exist between lexical choice and syntactic realization. The lexical choice strategy in GOSSIP is developed in detail in [Polguere 90]. This strategy is closest to the one used in ADVISOR II and presented in this chapter.

Divided between Content Planning and Syntactic Realization: In some cases, the structural decisions (*i.e.*, the building of the linguistic structure) are still performed before surface realization starts, but the paradigmatic decisions are delayed until the end of realization. In PENMAN [Bateman et al. 90], the realization system expects its input in a KL-ONE representation that must be a specialization of the *upper-model*. The upper-model is a set of basic ontologic distinctions that are required by NIGEL [Mann & Matthiessen 83b] to make linguistic decisions. For example, the upper-model distinguishes between the different types of events and processes that yield different clause structures, defining classes of semantic relations like `mental` or `attributive`. Since the semantic input to NIGEL needs to be a specialization of this upper-model, the linguistic structure must already be determined before realization starts. In particular, this means that the syntactic category (clause, nominal group or adjective) of each semantic constituent is determined in the input. For example, if the input contains a semantic relation of attributive, then an attributive clause is generated. If, however, the input contains an entity with some role like color, then a noun-group with a modifier is generated. In this sense, NIGEL's approach does not authorize cross-ranking realization - which consists in letting the realization component decide whether the same semantic modification of color is to be realized by an attributive clause or by a noun-group modifier.

For paradigmatic decisions, NIGEL accesses the lexicon only after the higher level syntactic decisions have been made. Sets of semantic features are used in each constituent where lexical items would occur and are sufficient for making syntactic choices (*e.g.*, determination of number, gender, agreement). Semantic features get added as the grammar systems make choices. After all syntactic choices have been made, the lexicon is accessed to replace each set of features with a lexical item. A lexeme may be preselected (by the deep generator for example) or directly chosen by the grammar (through a `lexify` realization statement). In the latter case, it would provide constraints on other choices. Systemicists term lexical choice as *the most delicate* of decisions as it is represented at the leaves of grammatical systems. In summary, NIGEL requires an input which is already linguistically structured and performs paradigmatic lexical choice at the end of the realization process.

During Content Planning and Before Syntactic Realization: Another class of generation systems positions the task of lexical choice as occurring somewhere during the process of deciding what to say, before the surface generator is invoked. This positioning allows lexical choice to influence content and to drive syntactic choice. Danlos [Danlos 87b] chooses this ordering of decisions for her domain. In her system, a *lexicon grammar* first selects lexical items for the predicative elements of the conceptual input. A *discourse grammar* is then used to combine clauses into valid discourse organization patterns which also determine syntactic realization (choice passive/active for example). Danlos's system thus performs content determination and lexical choice *before* discourse organization and syntactic realization (a very original position).

Rubinoff's IGEN [Rubinoff 92] also addresses the problem of the interaction between content decisions and linguistic decisions and implements an architecture that relies on explicit negotiation between the two components: the content planner requests a set of options from the lexical chooser to realize a certain conceptual element, and the lexical chooser replies with annotated options. The content planner selects among these annotations as it proceeds and combines the preferred options into an English message. Rubinoff designed a language of annotations that maintains a good separation of knowledge between the conceptual and linguistic components.

Other researchers advocate folding the lexicon into the knowledge representation. In this approach, as soon as a concept is selected for the text, the lexemes associated with it in the knowledge base would automatically be selected as well. This is the approach in KING [Jacobs 87] and FN [Reiter 90].

6.3.2. Lexicalization Strategy in ADVISOR II

The view taken in this chapter is that the lexicon provides the primary interface between a conceptual knowledge representation and the linguistic realm. The conceptual knowledge is represented in a linguistically independent way, *i.e.*, there is no notion of a linguistically motivated upper-model as in NIGEL. So, for example, in the ADVISOR II domain, the attribute `workload` of a class is useful for inferencing purposes (to warrant inferences like *the higher the workload of a class, the more time it requires from the student*). So a predicate `workload(class, value)` is used in the knowledge representation, even though this predicate does not fit easily under a single process type of the upper-model of NIGEL because the predicate `workload` does not correspond to a verb, and it can be realized by expressions of different categories: *the workload of AI is high* or *AI requires a lot of work.*

In the lexicalization method presented here, the lexicon provides resources to map conceptual relations onto linguistic predicative relations. The lexicon is therefore indexed by conceptual relations and entities. In general, lexicalization is performed before syntax. Within the lexical choice module, the flow of control is the following:

- An entry point in the conceptual network is selected.
- The entry point is lexicalized: a linguistic pattern corresponding to the conceptual node is selected. The linguistic pattern specifies a lexical element for the node **and** the new linguistic constituents that depend on it along with the mapping between these constituents and semantic elements.
- If more conceptual links are attached to the head and not covered by the lexicalization, consider whether they can be attached to the linguistic pattern as optional linguistic modifiers. If they cannot be attached, keep them on an agenda of conceptual relations to be covered.
- All new linguistic constituents (both obligatory members of the pattern and optional modifiers) are lexicalized in turn.

- When all linguistic dependents have been lexicalized, create a new linguistic constituent starting at one of the non-covered conceptual elements stored in the agenda. The new constituent becomes a new sentence. The process continues until all the conceptual network is covered.

The output of the lexicalization is an FD with a completely determined linguistic structure and all open class lexical items selected. So lexical choice is mostly performed after content selection and before syntactic realization, as in Meaning-text theory and in MUMBLE and TEXT.

6.3.3. Lexical Choice First, Syntactic Realization Second

Thus, in this approach, lexical choice is performed first before syntactic realization, in a pipeline architecture. The advantages of the pipeline approach over the integrated approach are clearly efficiency: if the two processes of lexicalization and syntactic realization can be separated, then the complexity of the pipeline approach is $l+s$ whereas the complexity of the integrated approach is $l \times s$, where l is the complexity of lexicalization and s the complexity of syntactic realization. The question is, therefore, how dependent are the two processes?

To prove that they are dependent, and that, therefore, the integrated architecture must be implemented, one must find cases where the same input constraint can be realized by lexical means in certain cases and by syntactic means in others. In addition, it must be shown that there are cases where it is necessary to switch from the lexical form to the syntactic form because it conflicts with other decisions.

In the ADVISOR II domain, I considered the following example as a possible case of inter-dependence between lexical and syntactic decisions: the conceptual relation to be conveyed is that a set of students took AI, with the argumentative evaluation that AI is difficult (attributed to the students and endorsed by the speaker). The lexical choice is between a conflation of the conceptual relation and the argumentative evaluation into a verb with connotation or a realization in two clauses:

Many of my students really struggled with AI.
Many of my students took AI and they found it really difficult.

The syntactic difference between these two versions is that the first one cannot be used at the passive voice, whereas the second one can:

** AI was really struggled with by many of my students.*
AI was taken by many of my students and they found it really difficult.

So, if the choice between active and passive voice is left to the syntactic component to satisfy focus constraints (as it is used in [McKeown 85], for example, to move the focus toward the beginning of the clause), then this syntactic decision conflicts with an earlier lexical decision.

The solution to this apparent violation of the independence between lexicalization and syntactic realization is to consider the choice of focus and the subsequent choice of voice as a lexical decision, *i.e.*, to consider the two patterns *X takes Y* and *Y is taken by X* as two distinct lexical entries. This is indeed the position taken by most recent lexically based theories of grammar (*e.g.*, LFG and HPSG). The two patterns, active and passive forms of *take*, are distinguished by using the focus constraint, which must be present at the time of lexicalization anyway to determine the perspective and the entry-point to the conceptual network. Under this view, there is no longer any conflict between lexicalization and syntactic realization. The lexicalization control presented here, therefore, differs from McKeown's dictionary interface [McKeown 85] which does not take into account focus information, nor any other pragmatic factors like stylistic constraints and argumentation during lexicalization.

So far, I have not found compelling evidence that would require revising the pipeline architecture between lexical choice and syntactic realization. I have, therefore, adopted the much more efficient and simpler serial architecture.

6.3.4. An Example

The control flow of the lexicalization procedure presented here is illustrated by the following example. Consider the input shown in Fig.6-6.

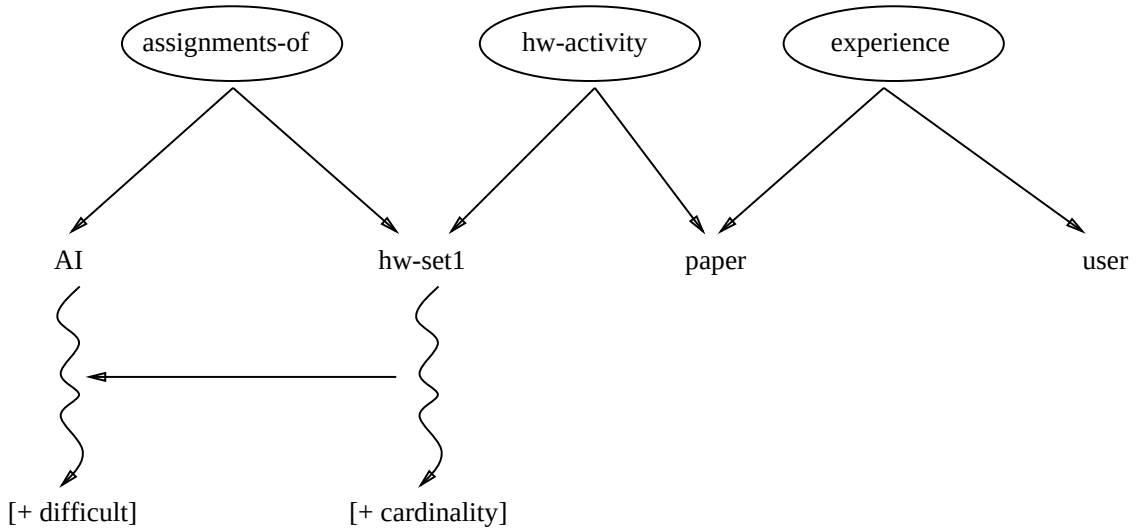


Figure 6-6: An input to the lexical chooser

An entry point is first selected. The network shown is a composite evaluation of the number of assignments in the AI class. So the perspective selected is the `assignments-of` relation, which covers the entities being evaluated. The choice of focus is constrained by the position of the utterance in an extended discourse. I assume here that the focus is set on the `AI` entity. The `assignments-of` relation is, therefore, lexicalized first as a clause, with the focus set on the argument 1. Three patterns are defined in the lexicon for this relation, corresponding to the following surface forms:

C has A
C requires A
In C, there is A

At this point, there is no constraint that allows choosing between these entries. So let's assume that the first option is selected. The lexical pattern identifies two new linguistic constituents, `possessor` and `possessed` in a linguistic relation of possession and it maps `possessor` to `AI` and `possessed` to the set `hw-set1`.

Since no more links reach the relation node `assignments-of`, the recursion proceeds to these two new constituents. The `AI` node is lexicalized first as an NP. The lexicon provides a choice between several names for `AI`:

Introduction to AI
Intro to AI
AI

The choice between these options is arbitrary, and the lexical chooser randomly selects *Intro to AI*. The `AI` node is the object of an argumentative evaluation on the scale of difficulty. This evaluation could be realized at this linguistic rank by adding a modifier to the NP *Intro to AI* as in *Intro to AI, which is difficult...* But the evaluation on `AI` is a piece of new information for the hearer (it is the conclusion of an argumentative inference) and, therefore, it cannot be presented as given, which is the effect of using a modifier of an element presented itself as given. Therefore other options must be selected to cover this argumentative evaluation at other linguistic ranks.

After the AI node, the second argument, the hw-set1 node is visited. This node is a set and participates in two conceptual relations plus one argumentative evaluation. When mapping the set description to an NP, one of the options is to embed the hw-activity relation as a modifier of the complex NP. The cardinality evaluation is realized by default by selecting a quantifier expressing the evaluation of the quantity. When these decisions are taken, the lexicalization of hw-set1 is: *(quantifier) assignments (modifier)* with the constraints that *quantifier* covers the cardinality evaluation and *modifier* covers the hw-activity relation.

The recursion, therefore, proceeds to these new nodes. *Quantifier* is a determiner whose semantic representation contains the evaluation of hw-set1. When mapped to a determiner, depending on the type of evaluation (absolute, relative or composite) certain features of the determiner are set. In this case, the realization features (realize-quantity evaluation) (orientation +) are set.

For *modifier*, the lexical chooser must decide whether the modifier is a describer (pre-modifier: noun or adjective) or a qualifier (post-modifier: relative clause or prepositional phrase). This depends on the relation hw-activity. Only certain relations can be realized as describers. In this example, the conditions for a describer are not met and a qualifier is selected. So the relation hw-activity is now traversed to be realized as a relative clause.

The conceptual relation hw-activity is mapped to the pattern *X consists of Y*, where *X* is mapped to the antecedent of the relative clause and *Y* is mapped to the node paper. After the paper node is lexicalized, the first two relations with their argumentative evaluations have been mapped to the sentence:

Intro to AI has many assignments which consist of writing essays.

At this point, the AI evaluation and the experience relation are still not covered. The experience relation could be attached to the NP *writing essays* but this would create an embedding of depth 2, which is rejected for stylistic reasons. So the relation needs to be realized in a distinct sentence. Finally, the AI evaluation needs to be realized. The choice is between realizing it at the level of the clause containing the realization of AI or higher up, by connecting its realization with the whole complex. One solution at the clause level is to choose a verb that expresses the evaluation of AI as a connotation. This is the case, for example, for the verb *to require*: *X requires Y* has a connotation that *X* is difficult. Another option is to add a resultative adjunct to the clause, *e.g.*, *AI has many assignments, making it a difficult class*. Politeness criteria, however, require that the qualification of AI as difficult be modalized. Neither of the clause-level realizations allow such modalization. So the lexical chooser ends up realizing the evaluation at the clause-complex level, using a connective to yield the final realization of the input:

Intro to AI has many assignments which consist of writing essays.
You do not have experience in writing essays.
So Intro to AI could be difficult.

6.3.5. Summary

The lexicalization procedure in ADVISOR II differs from all existing lexical choice methods by explicitly considering cross-ranking realization, *e.g.*, when deciding to realize an argumentative evaluation, all the sites which can be modified, from the clause-complex to the determiner level are explicitly considered and compared. In addition, this lexicalization procedure is the first to consider the choice of words for their connotations - viewing a pattern such as *X requires Y* as the conflated realization of two conceptual relations, include(X, Y) and difficult(X, +).

In terms of architecture, lexical choice and syntactic realization are serialized - *i.e.*, all lexicalization of open-class items is performed before any syntactic decision is made. Note of course that some paradigmatic decisions which do not affect the linguistic structure are still delayed until the syntactic realization stage, as discussed in Sect.6.4. In addition, closed-class items, in particular determiners and prepositions are selected at the syntactic realization stage. The most notable features of the method are that cross-realization is possible (allowing the same conceptual input to be realized at different levels of the syntactic tree), and that the lexical chooser is sensitive to pragmatic factors such as focus, argumentative orientation and politeness criteria.

I now provide a detailed account of the construction of linguistic constituents of category clause, NP and determiner, connectives and adjectives, with a special emphasis on the realization of argumentative evaluations. In each of the

following sections, I start by a review of the syntax of the constituent, then describe the semantic input required and discuss how the input is mapped to an appropriate syntactic structure.

6.4. Generation of Clauses

This section describes the lexical decisions made at the level of the clause. The underlying assumption is that the clause is governed lexically by a head verb - or, more generally, a process type. The clause is, therefore, structured mainly by selecting a process type. I first enumerate which configurations of conceptual elements can be mapped onto a clause. This defines the input to the lexical chooser. I then briefly review the features of the clause that the SURGE syntactic realization grammar expects to generate a clause. This defines the output of the lexical chooser. It is then possible to move on to a description of the lexical choice decisions made at the clause level. The mapping from the conceptual form to the syntactic form is performed in three stages:

- First, a *perspective* is selected in the conceptual network. The perspective is one relation in the network which is mapped to the head process of the clause. Other relations in the conceptual network are then either attached to the head clause as modifiers of its arguments or as subordinate clauses. The pragmatic factors controlling the decisions involved in this first structuring step involve focus of attention and style criteria.
- Second, the head of the clause is lexicalized: a lexical entry is selected. Since lexical entries for verbs specify their subcategorization frame, it becomes possible to map the conceptual arguments of the head relation to syntactic arguments: either participants or circumstantials. At this stage, I also show how an argumentative floating constraint can be merged into the head process by selecting a verb with a connotation.
- Finally, the mood and modality of the clause are derived using information about the position of the clause in the emerging syntactic structure and politeness constraints.

The main innovation of this approach is to show (1) how lexical choice is controlled by pragmatic factors (focus, argumentation, politeness), (2) how several conceptual elements can be mapped onto the same linguistic constituent (a clause realizes a conceptual relation plus an argumentative evaluation), and (3) how the same conceptual category (relations) can be realized by clauses, prepositions, nouns or adjectives. This last point is further developed in Sect.6.5 describing the lexicalization of NPs.

6.4.1. Conceptual Elements Realizable by a Clause: Input

I first enumerate which semantic elements can be realized by a clause. The next section discusses how each one of these elements is mapped from the conceptual level to the clause level. The complete list of conceptual elements is (cf. Sect.6.2):

- individuals
- sets
- relations
- argumentative evaluations

The following configurations of input conceptual elements can be realized by a single clause:

- A single relation: *AI has 6 assignments.*
- A single argumentative evaluation: *AI is difficult.*
- A relation with an argumentative evaluation in a composite clause: *Logic makes AI difficult.*
- A relation with an argumentative evaluation using a verb with connotation: *AI requires many homeworks.*
- A relation with an argumentative evaluation using an argument carrying the evaluation: *AI has many assignments.*

In addition, many more combinations can be covered by a clause with circumstantials (basically, each circumstantial covers a conceptual relation), but these additions generally do not affect the choice of the lexical head of the clause, and I have not studied how circumstantials are generated from a conceptual representation in the lexical chooser.

So the lexical chooser looks for one of the listed combinations and matches it with some entry in the lexicon. Note that I make here the assumption that the conceptual representation is of finer-grain than a clause: several conceptual relations can combine into a single clause, but I do not consider the case where several clauses realize a single conceptual relation. I focus in this work on the merging of conceptual relations with argumentative evaluations into a single clause to illustrate the non-isomorphic relation between the conceptual structure and the linguistic structure. As suggested in Sect.4.1.3 (p.88), there are many other cases where several semantic elements get conflated and realized by a single linguistic constituent.

```
(def-test input1
  "You do not have experience in one assignment
   which Intro to AI requires."
  ((cat topos)
   (left
    ((evaluated {justification assignments})
     (scale ((name "cardinal")))
     (orientation +)
     (type composite)
     (value {right})))
   (right
    ((evaluated {justification class})
     (scale ((name "difficult")))
     (value +)
     (orientation +)))
   (justification
    ((rule eval-12)
     (assignments
      ((cat set) (index hw2) (kind ((cat assignment)))
       (cardinality 2)))
     (class ((cat class) (index ai1) (name ai-class)))
     (hw-activities
      ((cat set) (index hwa1) (kind ((cat hw-activity)))
       (cardinality 1)
       (realize-extension yes)
       (extension
        ((car ((semr ((cat hw-activity) (name paper1)))))
         (cdr none)))))
     (class-relation
      ((name assignments)
       (1 {justification class})
       (2 {justification assignments})))
     (user-relation
      ((name experience) (orientation -)
       (1 ((cat student) (name hearer)))
       (2 {justification hw-activities})))
     (object-relation
      ((name hw-type)
       (1 {justification assignments})
       (2 {justification hw-activities})))))))
```

Figure 6-7: A conceptual input in FUF notation

Figure 6-7 shows an example of input configuration sent to the lexical chooser. This form is generated by an evaluation function (cf. Chap.7) and corresponds to a composite evaluation of the AI course. A literal realization of this evaluation is *you do not have experience in writing papers, which one assignment of Intro to AI requires; so AI is difficult for you.* Note that the information extracted from the knowledge base is encoded under the feature `justification` and consists of three relations, under the features `class-relation`, `user-relation` and `object-relation`. The elements which can be realized by a clause in this input are: the evaluation under `left`, the evaluation under `right`, the three relations under `{justification}`, and combinations of these five.

6.4.2. Summary of the Syntax of the Clause: Output

As described in Sect.5.2, the following features are expected by SURGE to generate a clause:

- A process of type either simple, composite or lexical. If the process is lexical, its lexical head must be specified, otherwise, for simple and composite processes, default verbs are selected by the grammar. These defaults can be overridden if desired.
- A configuration of roles corresponding to the type of the process.
- The mood feature and if necessary the scope feature pointing at one of the roles.
- The modality feature.
- The focus feature pointing at one of the roles.

Note that this input does not correspond to the conceptual input presented in Sect.6.2. It is more syntactic in nature, even though it hides a good part of the surface syntax of the clause. In terms of the Meaning-text theory, this input corresponds to the *deep-syntax representation* layer. The important distinction is that the semantic terms used at this level are part of a *linguistic semantic*: notions like agent, affected, ascriptive, etc. do not correspond to the conceptual relations of the domain: they are derived from a generalization over linguistic observations and linguistic structures. The role of the lexical chooser is to map conceptual relations onto this form of linguistic semantic. There is a choice point at this level which cannot be avoided: for example, the conceptual relation `assignments-of` which links a course to a set of assignments can be realized by a linguistic relation of possession *AI has 6 assignments* or by the lexical relation *require*, which I represent as a lexical process. These two relations are very different, but in the ADVISOR II domain, they can both realize the same conceptual relation.

So there is no point in forcing the conceptual relation as a specialization of the possessive relation in the knowledge base, or of the `requirement` relation. Instead, I leave this mapping to the lexical chooser and do not view it as a conceptual decision, but simply as an ‘‘arbitrary’’ - *i.e.*, unexplained - expression decision. The fact that a syntactic form of possession is used in an expression does *not* mean that a conceptual possession is denoted.[52] Note that this position is in sharp contrast with the upper-model approach of PENMAN [Bateman et al. 90] and with the completely knowledge-based (*i.e.*, conceptual) approach of Jacobs [Jacobs 87] or Reiter [Reiter 90].

6.4.3. Choosing a Perspective: Clause Planning

Clause planning is the first stage of the clause generation. It consists of mapping the conceptual network to a hierarchical linguistic structure, where constituents are attached to heads. The first step of clause planning is the selection of an entry point in the input, which I call the *perspective*. When generating a clause, the entry point must be a relation. In the example in Fig.6-7, the entry points can be one of the three relations under `justification`, yielding the following realizations:

- **Perspective on user-relation:** *You do not have experience in writing essays which two assignments of Intro to AI require.*
- **Perspective on object-relation:** *Two assignments of Intro to AI require writing essays, in which you do not have experience.*
- **Perspective on class-relation:** *Intro to AI requires two assignments which require writing essays, in which you do not have experience.*

Focus information can be used to constrain the perspective selection. Assuming that the focus is maintained across the discourse using thematic progression rules (which I have not implemented) the following rule is used to determine the perspective using focus as an input parameter: the perspective in a network of conceptual relations is a relation which contains the focus as one of its arguments. For example, in the previous example, assuming the focus is on the AI class, only the `class-relation` can serve as a perspective, resulting in the third sentence shown

[52]Of course, a conceptual possession is most of the time realized by a syntactic form of possession. I only reject the reverse inference.

above. If the perspective is on writing papers, then either `object-relation` or `user-relation` can serve as perspective. Without further information on the communicative structure of the input, there is no way to choose between these two options. So the choice here would be arbitrary.

Thus, the notion of perspective can be understood as a form of focus on relation as opposed to the standard usage of focus on entity (as used for example in [McKeown 85]). While focus has been used in previous work to constrain syntactic realization (choice of active vs. passive voice, selection of *it*-cleft), the choice of perspective affects lexical choice, by determining the selection of the main verb and constraining clause planning.

In summary, the perspective selection rule has reduced the choice of perspective and focus to the choice of focus only. I assume a discourse focus tracking algorithm exists which determines which semantic element is in focus before generating each clause, and how the focus switches coherently as discourse progresses. As the notion of perspective is more general than that of focus, more pragmatic factors are likely to be required to select the perspective in a complex network. An operational notion of relevance is likely to be useful to derive a more general perspective selection procedure.

The second step of the clause planning process consists of attaching other conceptual elements as modifiers of the head relation. This mapping is performed in two steps. Assume the focus is on AI, and the perspective is on the class relation. First, the head constituent of the linguistic structure is built from the description of the class relation. This is performed by the following fragment of the grammar:

```
;; Construct the head of the linguistic realization of the network
(head
 ((cat clause)
  ;; This is a clause realizing the class-relation
  (semr {^2 class-relation})
  ;; The head of the clause is a process that needs to be lexicalized
  (process ((semr ((cat class-relation)
                   (polarity {^4 class-relation orientation})
                   (name {^4 class-relation name})))))
  ;; It has two participants, realizing each of the arguments of
  ;; the class-relation.
  (roles ((1 ((semr {^3 class-relation 1})))
          (2 ((semr {^3 class-relation 2}))))))))
```

The mapping described by this fragment is also depicted in Fig.6-8. This mapping illustrates two points:

1. The linguistic structure is built out of the conceptual structure, and each linguistic constituent contains a feature `semr` pointing to the conceptual element it realizes (`semr` stands for semantic realization).

2. The argument structure of the clause is not specified in terms of participants or lexical roles because the clause has not yet been lexicalized. The only decision made so far was to realize a relation by a clause - the head of the clause has not been determined. When it is selected, the generic `roles` feature will get mapped to the appropriate argument structure.

The second step of the mapping is to map the other relations onto modifiers of the arguments of the head clause. In the example, this is achieved by the following grammar fragment:

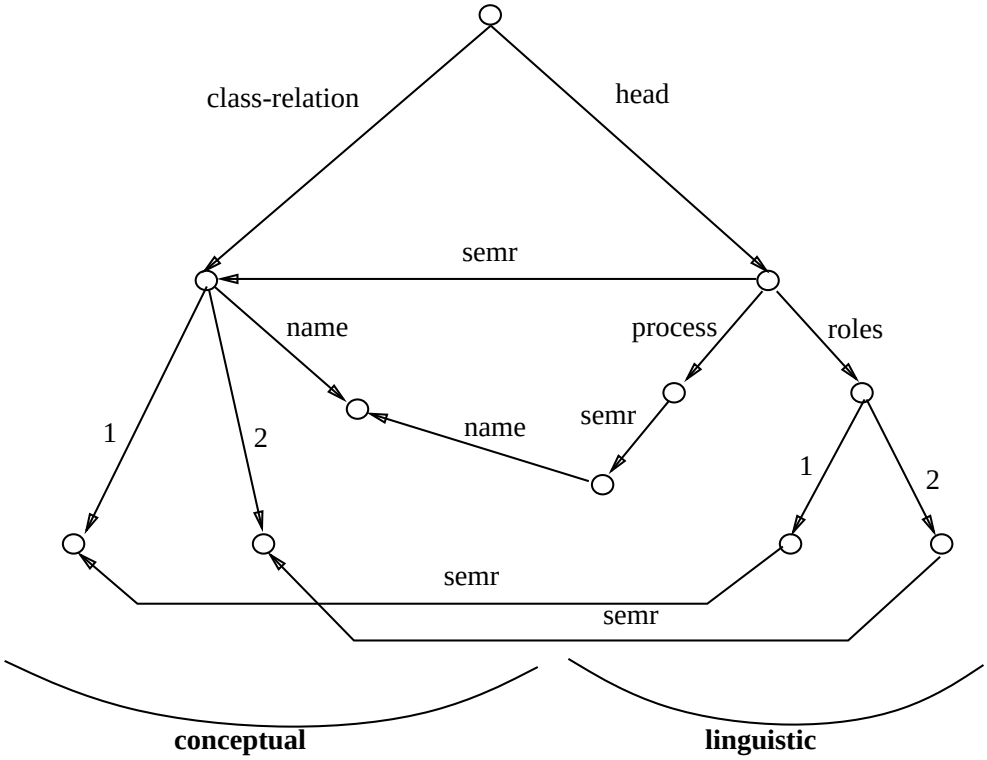


Figure 6-8: Construction of a clause structure

```
;; Object relation shares an argument with head
(object-relation ((1 {^2 head semr 2})
                  (covered yes)))
;; Make object-relation a modifier (intension)
;; of the common argument under head
(head
 ((roles
    ((2 ((semr
          ((cat set)
           (realize-intension yes)
           ;; The shared argument 1 in the embedded relation is
           ;; replaced by an indefinite variable (name none)
           ;; and becomes the value of argument.
           (intension ((process ((name {^7 object-relation name})))
                       (roles ((1 ((semr ((name none)
                                          (cat {^5 kind cat})))))
                               (2 ((semr {^8 object-relation 2})))))
                       (argument {^ roles 1})))))))))))
```

The `object-relation` is mapped onto the intension slot of the role 2 of the head clause, when it is found that the object-relation has for argument 1 the argument 2 of the class-relation. This mapping is illustrated in Fig.6-9. Note that the `intension` feature is the modifier slot of an individual of type set. The intension description has the same format as a clause in the linguistic structure. This does not mean that the intension will be realized by a clause - it can be realized by an adjective or a noun - but I view these modifiers as derived from the relative clause construction using only linguistic derivations, following [Levi 78]. The realization of the set category is described in the next section. For now, I only show how the structure of the arguments is computed once the perspective is selected.

Another option to attach relations of the conceptual network which are not selected as head to the head clause is to attach them as subordinate clauses of the head clause. This option is selected when embedding the clause would lead to a modifier too deeply embedded. Stylistic criteria determine what depth of embedding is acceptable. In the

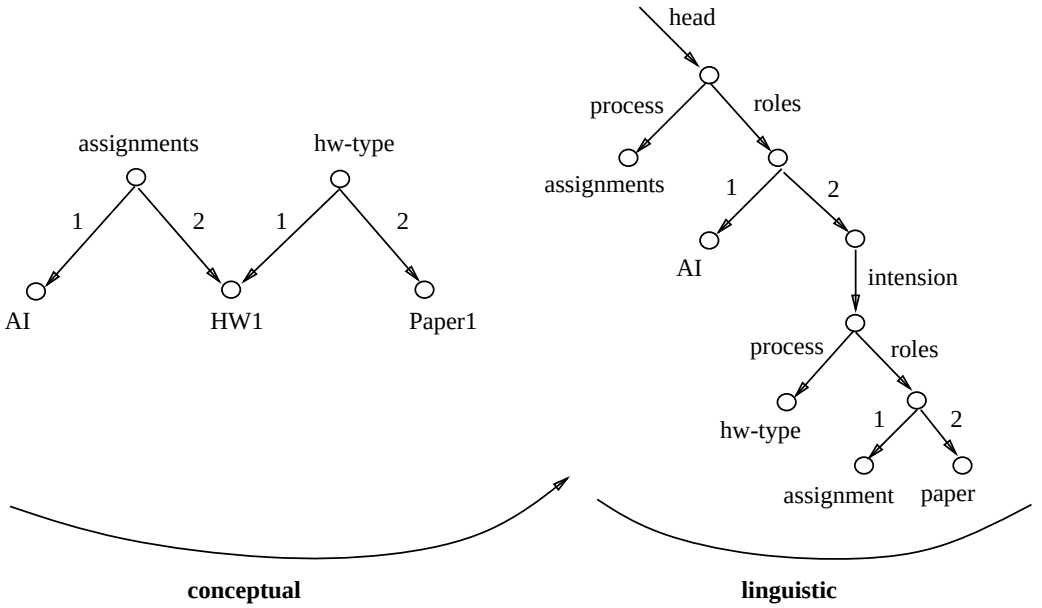


Figure 6-9: Mapping of a relation as a modifier

current implementation, one-level of embedding only is accepted. Thus, the clause combination (1) is preferred over the embedded combination (2) because (2) has a relative clause embedded at level 2:

(1) Intro to AI has many assignments which consist of writing essays. You do not have experience in writing essays.
(2) Intro to AI has many assignments which consist of writing essays in which you do not have experience.

More complex stylistic criteria could be used to control this decision (balance among the constituents, number of words etc).

In summary, the first step of the mapping from conceptual network to clause is (1) to select a perspective among the conceptual relations of the network, which determines a head clause, and (2) to attach the remaining relations as either embedded or subordinate modifiers of the head clause. The perspective is selected using focus constraints; the choice between embedding or subordination is based on simple stylistic criteria. The output of this stage is a hierarchical structure where heads correspond to linguistic constituents of a set category (clause or NP), but where the lexical heads are not yet selected. These two operations can be seen as part of a form of *clause planning*, similar to text planning at the paragraph level. In the next section, I also describe the operation of *NP planning*, which consists of choosing a head in a description, and mapping modifiers to the different forms of syntactic modifiers an NP contains. These operations of sub-sentence planning are possible in this approach because the conceptual input is not already linguistically structured. Such planning is a major source of paraphrasing power, and since it is controlled by pragmatic factors, it also increases the sensitivity of the generator.

6.4.4. Building the Clause Argument Structure

After the conceptual network has been turned into a hierarchy of linguistic constituents, the heads of this hierarchy must be lexicalized. As noted above, the clause description shown in Fig.6-9 contains a generic description of the argument structure: it is neither a set of participants under a known transitivity process type, nor a set of lexical roles subcategorized by a known lexical head. In addition, the head of the clause (the `process` feature) is not yet lexicalized. The final semantico-linguistic structure which serves as input to SURGE is built when a lexical entry is selected to realize the `process` feature. The lexical entry specifies how to map the generic `roles` feature to either a set of participants (if the selected process is a specialized transitivity process) or to a set of lexical roles (if the selected process is a lexical entry with its own specific subcategorization frame).

I therefore describe in this section the lexical resources available when lexicalizing the head of a clause. The lexicon is indexed by conceptual relations: for each conceptual relation, the lexicon lists the items which can realize it in different contexts.

Four types of lexical resources can be used to lexicalize the head relation:

1. A simple transitivity specialization.
2. A simple lexical entry.
3. A composite transitivity specialization.
4. A composite lexical entry expressing a connotation.

Transitivity specializations are selected when the conceptual relation can be expressed by one of the transitivity configurations enumerated in Sect.5.2.1. In the other cases, a specific lexical item with its subcategorization frame is selected.

A crucial feature of this lexicalization procedure is that it allows several conceptual relations to be covered by a single lexical entry. Two such cases occur at the clause level: using a composite transitivity specialization and using a lexical head conveying a connotation. In this work, I focus on the merging of configurations of two relations - one conceptual relation and one argumentative evaluation. This illustrates how lexical choice at the level of the main verb of the clause can serve as a site to realize floating argumentative constraints.

Figure 6-10 shows a fragment of the grammar dealing with the lexicalization of the relations which are specializations of `course-relation`. Those relations are represented by an FD containing the following configuration:

```
((head ((process ((semr ((cat course-relation) ...))))
        (roles ((1 ((semr ...)))
                (2 ((semr ...)))))))))
```

That is, after clause planning, the head clause has been selected and placed under the `head` feature. The process of this clause realizes a conceptual relation of category course-relation. The generic roles of the clause have been mapped to conceptual arguments, and are represented under features `{roles 1}` and `{roles 2}`. The grammar shown is unified with the constituent appearing at level `{head process semr}`. I show four branches in this alt covering each one of the four types of lexical resources available at the main verb level:

- **Simple transitivity process:** The `area` relation is lexicalized using a specialization of the ascriptive process type in the simple transitivity system. Example lexicalizations are *Compilers is a software class* or *Analysis of Algorithms is theoretical.* In this case, the simplest case of lexicalization, the generic roles of the head clause are mapped onto the participants of the clause: carrier and attribute.
- **Lexical process:** The `topics` relation is lexicalized using a simple lexical entry with its specific subcategorization frame. An example of lexicalization is *AI deals with NLP and Knowledge-representation.* The lexical entry for the `topics` relation is shown in Fig.6-11. In this case, the generic roles are mapped onto the lexical roles identified by the `subcat` feature of the main verb. For example, when the selected main verb is *X deals with Y*, the generic role 1 is mapped to the lexical role X of *to deal with*, and the generic role 2 is mapped to the NP Y of the PP *with Y*.
- **Composite transitivity process:** The third branch covers all instances of the course-relations category, and uses a composite process type from the composite transitivity system. Here, the course-relation and an argumentative evaluation (when present, as checked by the `(ao given)` feature) are merged into a single clause pattern of the form *Logic makes AI difficult* or *Software Design is made time-consuming by a programming project.* The argumentative evaluation is mapped to the ascriptive component of the composite process, while the course relation is mapped to the material component: the class is mapped to both the affected and the carrier roles, the object of the course relation (role 2) is mapped to the agent role, and the argumentative scale is mapped to the attribute role.
- **Lexical process with connotation:** Finally, the fourth branch covers the `assignments` relation, which

```
(def-alt course-relation (:index name)
  ;; For each type of course-relation, determine possible
  ;; lexicalizations and their cat and identify sub-constituents.

  (
   ;; Simple transitivity specialization
   ;; Lexicalization: C is A
   ((name area)
    (lex-cset ((+ {^3 roles 1 semr} {^3 roles 2 semr})))
    ({^} (({^ roles 1} ((kind ((cat class) (typical-size 5)))))
          ({^ roles 2} ((kind ((cat area) (typical-size 1)))))
          (type ascriptive)
          ({^ participants} ((carrier {^2 roles 1})
                             (attribute {^2 roles 2})))))))

   ;; Simple lexical entry
   ;; Lex: Course <deal with, cover, include, involve> Topics
   ((name topics)
    (lex-cset ((+ {^3 roles 1 semr} {^3 roles 2 semr})))
    ({^} (;; The constituents are class and topics under roles.
          ;; Typically 1 class covers 3 topics.
          ;; Typically 1 topic is covered by 1 class.
          ({^ roles 1} ((kind ((cat class) (typical-size 1)))))
          ({^ roles 2} ((kind ((cat topic) (typical-size 3)))))
          (:& topics))))

   ;; Composite transitivity
   ;; Object makes Course Attribute.
   ;; The composite type covers both the relation and the AO.
   ;; It does not depend on the specific relation class/object.
   (({^ ao} given)
    ({^ ao} ((evaluated {^2 roles 1})
             (covered yes)))
    ({^} ((type composite)
          (relation-type ascriptive)
          ({^ participants} ((carrier {^2 roles 1})
                             (attribute ((semr {^2 ao scale})))
                             (agent {^2 roles 2})
                             (affected {^ carrier})))))))

   ;; Composite transitivity (give) and Connotation (require)
   ;; Lexicalizations: Course <require, have> Assignment,
   ;; There is Assignment in Course
   ;; Teacher gives A in C.
   ((name assignments)
    (lex-cset ((+ {^3 roles 1 semr} {^3 roles 2 semr})))
    ({^} (;; The constituents are class and assignments
          ({^ roles 1} ((kind ((cat class) (typical-size 10)))))
          ({^ roles 2} ((kind ((cat assignment) (typical-size 4)))))
          (:! assignments))))

    ;; ... more course relations ...

    ))
```

Figure 6-10: Lexical entries for the course relations

```
(def-conj topics
  ;; Mapped to a process of type lexical
  ;; with arguments in lex-roles.
  (type lexical)
  ({^ roles} {^ lex-roles})
  (ralt topics (:demo "Lexicalization of topics relation")
        (((lex "deal")
          (subcat ((1 {^3 roles 1})
                   (2 ((cat pp)
                       (prep ((lex "with")))
                       (np {^4 roles 2})
                       (np ((realization {^2}))))))))
         ((lex ((ralt ("cover" "include" "involve"))))
          (subcat ((1 {^3 roles 1})
                   (2 {^3 roles 2})))))))
```

Figure 6-11: Lexical entry for the relation `topics`

can be lexicalized by a lexical entry with a connotation: *AI requires many assignments.*[53]

The lexical entry for the `assignments` relation is shown in Fig.6-12. The first branch of the alt describes the verb *to require* as a process of type lexical, specified with its subcategorization frame. In addition, `require` has an argumentative connotation: *C requires X* is used when *C* is positively evaluated on the scale of difficulty, as indicated by the AO feature in the lexical entry. This entry can be used to merge an argumentative evaluation of the class on the scale of difficulty with the conceptual relation of assignments.

Note that the type of connotation described in the lexical entry for *to require* is very close to the lexical presuppositions used, for example, in [Fillmore 71], [Ducrot 72] and in [Polguere 90, pp.236-249]. For comparison, examples of definitions for the verbs of judging studied by Fillmore are shown for comparison in Fig.6-13. The lexical description used in this work does not distinguish between asserted and presupposed elements explicitly, but such multi-level semantic descriptions of lexical items confirm the need for a lexicalization strategy capable of merging several conceptual relations into a single clause. In the description presented in this section, the argumentative evaluations are floating semantic elements which can be mapped onto different structural sites in the clause structure. One of those sites is the lexical head of the clause.

In summary, I have shown in this section the form of the lexicon used to map a conceptual network (after it has been put in hierarchical form by the clause planning step) onto a linguistic clause structure. The clause structure is characterized by a process type and a set of participants (or lexical roles depending on the process type). Lexical entries specify (1) what is the word used for the main verb of the clause and (2) how the generic arguments of the conceptual network are to be mapped onto linguistic arguments in the subcategorization frame. Four types of lexical entries have been identified: simple transitivity processes, simple lexical processes, composite transitivity processes and lexical processes with connotation. The later two types of processes can be used to merge into the main verb of a clause the realization of both a conceptual relation and an argumentative evaluation. When such entries can be selected, *i.e.*, when a matching combination of conceptual relation and argumentative evaluation are part of the input, the main verb becomes an eligible site for the expression of the argumentative floating constraint.

---

[53] Two other simple configurations can be used to lexicalize the assignments relation:

- Simple process type: *AI has 6 assignments* (possessive)
- Simple process type with circumstantials: *There are 6 assignments in AI* (existential plus location circumstantial).

The last two branches in Fig.6-12 correspond to these simple process types. In the last branch, one of the core arguments of the conceptual relation is mapped onto a circumstantial in the clause - showing once more that the conceptual structure and the linguistic structure are not isomorphic.

```
(def-alt assignments (:demo "Lexicalization of assignments")
  (
   ;; C requires A: type lexical
   ;; with connotation (=> + difficult)
   ((type lexical)
    (lex "require")
    ({^ roles} {^ lex-roles})
    ({^ ao} given)
    ({^ ao} ((evaluated {^2 roles 1})
             (scale ((name "difficulty")))
             (orientation +)
             (covered yes)))
    (subcat ((1 {^3 roles 1})
             (2 {^3 roles 2}))))

   ;; C has A
   ((type possessive)
    ({^ participants} ((possessor {^2 roles 1})
                       (possessed {^2 roles 2}))))

   ;; In C, there is A.
   ((type existential)
    ({^ participants} ((located {^2 roles 2})))
    ({^ circumstances} ((in-loc {^2 roles 1}))))))
```

Figure 6-12: Lexical entry for the relation `assignments`

```
ACCUSE [judge, Defendant, situation (of)]
        SAY[Judge, X, addressee]
Meaning: X = responsible[situation, defendant]
Presupposition: Bad(Situation)

CRITICIZE [affected, defendant, situation (for)]
        SAY[judge, X, addressee]
Meaning: X = Bad(situation)
Presupposition: Responsible(defendant, situation)
Presupposition: Actual(situation)

EXCUSE [defendant, situation]
        SAY[defendant, X, addressee]
Meaning: X = Not(Responsible(defendant, situation))
Presupposition: Bad(Situation)
Presupposition: Actual(Situation)
```

Figure 6-13: Semantic descriptions for verbs of judging.
From [Fillmore 71, pp.282-286]

6.4.5. Choosing a Mood and a Modality: Politeness Constraints

Recall from Sect.6.4.2 that the output of lexicalization at the clause level must contain the following information to provide the necessary input to SURGE:

- A process description and corresponding role structure.
- The mood feature and if necessary the scope feature pointing at one of the roles.
- The modality feature.

- The focus feature pointing at one of the roles.

The previous two sections have shown how to select the process description and the corresponding role structure. In addition, I assume that the focus feature is provided in input to lexicalization and is maintained by a thematic development module (not currently implemented).

The last elements, then, that need to be derived from the conceptual structure are the mood and modality of the clause. Both of these features are related to the inter-personal situation: how the speaker relates to the hearer. The first aspect of this relation, is to determine the function of the clause being produced: provide or query services or information. In this work, I have restricted the scope of the system to answering questions; that is, providing information. So the mood is in general set by default to declarative. If the clause being generated is embedded, then the mood can be set to either relative (or one of its specializations) or to bound. These decisions are made by the syntactic realization grammar, based on the structure of the input produced by the lexical chooser, so no special treatment must be made by the lexical chooser to set the mood.

To set the modality, the system must take into account two factors: how certain the information it provides is, and how it affects the hearer. In the ADVISOR II system, information about the hearer is stored in a user model, and I assume that it is either obtained by direct interrogation or by inference, using the plan inference techniques described in [McKeown 88]. Information that is inferred either through these techniques, or through argumentative inference (*i.e.*, by chaining through topoi), is marked to be expressed with the modality of inference. All the rest is marked to be expressed with the modality of fact.

The inference modality can be expressed by several modals:

AI covers NLP. It must interest you.
AI has a lot of programming. It should interest you.
AI has many assignments. It could be difficult.

I view the difference between *must* and *should* as a difference in degree - *must* being the mark of a stronger inference than *should*. The difference between *should* and *could* in this context seems to be related to a phenomenon of politeness, as identified in [Brown & Levinson 87, Sect.5.4 on negative politeness]. The notion of politeness used in this work is based on three factors, called *Power* (relative power of speaker and hearer), *Distance* (social distance) and *Ranking* (ranking of imposition in the particular culture) [Brown & Levinson 87, p.74]. The theory maps these three inter-personal parameters to the choice of a linguistic *interaction strategy*. Interaction strategies listed in [Brown & Levinson 87] include `be indirect`, `hedge` or `seek agreement`. The role of the interaction strategies is to manage a trade-off between the conflicting requirements of two fundamental pulsions in social interaction [Brown & Levinson 87, p.62]:

- *Negative Face:* the want of every 'competent adult member' that his actions be unimpeded by others.
- *Positive Face:* the want of every member that his wants be desirable to at least some others.

The output of this theory is the selection of some linguistic tools to express a face-threatening act (FTA) in a situation characterized by a triple (P,D,R). In the ADVISOR II context, deriving the conclusion that a course is difficult for the student is a face-threatening act: it impedes the student's desire to take the course. Therefore, some of the negative politeness techniques listed in [Brown & Levinson 87] apply: hedge, dissociate speaker and hearer from the expression by avoiding the personal pronouns *I* and *you*, use impersonal forms and nominalizations. The politeness value of these techniques explains why the forms *you would find it difficult*, *it should be difficult for you* and *it could be difficult* are more and more ''polite'' in the sense that they express more and more distance between the hearer and the expression.

I have partially implemented this idea in the region of the grammar expressing argumentative evaluations as clauses. The idea is to associate two pieces of information with each argumentative evaluation: whether it is a *generic* evaluation or a *specific* one; and whether it is an FTA or not. A generic evaluation does not use any information specific to the hearer in its justification; a specific one relies on hearer-specific information. For example, an evaluation function infers that a course is difficult when it has more than 6 assignments. This is a generic evaluation. Another evaluation function infers that a course requiring assignments of a type in which the hearer has no experience is difficult. For example, for a student who has no experience writing essays, AI can be difficult

because it requires writing two essays. This is a specific evaluation. The second classification of evaluations is between FTAs and non-FTAs: an FTA is an evaluation which impedes or contradicts one of the goals of the hearer. Given the choice between a generic and a specific evaluation which implies an FTA, the planning system selects the generic one to fulfil one of the politeness strategies of dissociating the hearer from the FTA. If only a specific evaluation is available, then its *realization* is made generic if possible, by avoiding using a form where the pronoun *you* is used. In addition, a distancing modal is used: *could* expressing a possibility is preferred over *should* expressing an inference.

The politeness constraints identified by Brown and Levinson can be used more extensively in the lexical chooser. In particular, the interaction strategy to prefer nominalizations and impersonal forms could be used during clause planning. Further investigation in this connection seems a promising area of future work.

In summary, the mood of all clauses is set by default to declarative in the simplified context of the ADVISOR II system. For embedded clauses the mood is set by the syntactic realization module based on the structural position of the clause. Modality is set by default to fact, except in the case of argumentative evaluations realized by clauses. In this case, depending on whether the evaluation is generic or specific and whether it is an FTA or not, a modality of inference or possibility is used.

6.4.6. Conclusion: Constraints on Lexicalization at the Clause Level

The lexical decisions made when constructing a clause are:

- Choose a conceptual relation to be the head of the linguistic structure.
- Map the head relation to a clause structure, with generic arguments 1 and 2.
- Map connected relations as modifiers of the arguments of the head clause.
- Lexicalize the head relation and build the argument structure corresponding to this lexical choice. Possibly, merge compatible argumentative relations with the lexical head if an appropriate composite process is found.
- Choose a mood and a modality for the clause.

These decisions rely on the following information in the conceptual input and in the lexicon:

- The focus node is identified in the input. This choice imposes the selection of the perspective (head relation) and the voice selection at the syntactic realization stage. If the focus is not provided by a discourse model, I assume that the target of the argumentative evaluation is the focus of the clause.
- The lexicon contains entries indexed by conceptual relations and mapping a conceptual relation to a linguistic argument structure. It also contains composite entries which can merge the relation plus an argumentative relation into a single clause.
- The modality selection of clauses realizing an argumentative evaluation is constrained by two factors: whether the evaluation is specific or generic; and whether the evaluation is a face-threatening act or not.

The main contributions of this lexicalization technique at the clause level are:

- **Clause planning**: A separate stage called *clause planning* has been identified in the lexicalization process, where the head of the linguistic structure is identified, and non-head conceptual elements are attached to the head in different ways (as arguments, modifiers or subordinate clauses) and the syntactic categories (clause, NP) of the head and its attached constituents are determined. This clause planning step is flexible (many options are considered) and sensitive to pragmatic factors (focus, style, argumentation and politeness). It therefore significantly enhances the fluency of the generator (by increasing its paraphrasing power and its pragmatic sensitivity).
- **Merging of floating constraints:** A single linguistic clause can realize a single relation or a combination of two conceptual elements - one clause plus one argumentative evaluation. This contributes to fluency by increasing compactness. It also shows how paradigmatic lexical selection can be constrained by argumentative constraints.

- **Cross-ranking:** The same conceptual material - relations - can be realized by clauses, nouns or adjectives (when relations are mapped to modifiers of an NP by the clause planner).
- **Pragmatically sensitive lexical chooser:** The mapping from conceptual to linguistic structure is a complex transformation, taking into account focus, stylistic, argumentative and politeness constraints, instead of a simple isomorphic mapping as is often the case in generators using an input already linguistically structured.

6.5. Generation of NP

This section describes the lexical decisions made at the level of the noun group. Noun groups occur mainly as the arguments of clauses. The syntax of the English noun group is much less constrained than that of the clause. I first describe the conceptual representation of elements which can be realized as NPs, emphasizing the representation of sets. I then summarize the list of features that SURGE, the syntactic realization grammar, expects as input to properly generate an NP. This list of features defines the output of the lexical chooser at the NP level. I then describe how the input conceptual description is mapped onto this list of syntactic features. The task of *NP planning* is particularly important at the NP level because the syntax of the NP distinguishes between four types of syntactic modifiers (in addition to the determiner sequence) and the NP planner component of the lexical chooser must map each conceptual modifier to an appropriate syntactic modifier. The NP planning process is described in two steps: first, a high level structure is built, which in particular identifies the head of the NP (Sect.6.5.3). Second, the conceptual modifiers are mapped to syntactic modifiers (Sect.6.5.4).

6.5.1. Conceptual Elements Realizable by an NP: Input

In this subsection, I enumerate which conceptual elements can be realized by a noun group as a whole. This constitutes the input to the lexical chooser. The next subsection presents the output of the lexical chooser, *i.e.*, the list of syntactic features expected by SURGE to produce an NP.

The following configurations of input conceptual elements can be realized by a single NP:

- An individual.
- A set.
- A set modified by some conceptual relations.
- Any of the above configurations combined with an argumentative evaluation.

Sets are the only category in this list which is not directly derived from knowledge-base objects. I therefore briefly describe how sets are encoded in the input to the lexical chooser, and how set descriptions are derived from the knowledge-base.

I use the following features to represent sets at the semantic level:

- Extension
- Cardinality
- Kind
- Intension
- Reference

The extension feature lists the elements of the set if they are known. It is a list of individuals. The cardinality is a number, which is not always known. The kind is used when the set is a homogeneous set - every element is of the same kind. The kind is a class in the knowledge representation model. Two non-extensional features are used: reference and intension. Intension is a relation and reference is recursively a set description. Not all sets need to have an intension or reference specified. The logical definition of a set described by these features is the following:

$S=\{x \in \text{Reference} \mid \text{Intension}(x)\}$

An example FD description for the following set is shown in Fig.6-14:

(E) $S1=\{x \in \text{TOPICS} \mid \textit{Interest}(x,\textit{student}) \wedge \textit{Area}(x,\textit{AI})\}$

Intuitively, this set contains the 7 topics that are of interest to the user among the 10 topics which are covered in AI.

A set of 7 topics which:
(1) are among the 10 topics covered in AI
(2) interest the student.

```
((cat set)
 (kind ((cat topic)))
 (cardinality 7)
 (reference ((cat set)
             (kind ((cat topic)))
             (cardinality 10)
             (intension ((cat class-relation) (name area)
                         (argument {^ 1})
                         (2 ((cat field) (name AI)))))))
 (intension ((cat user-relation) (name interest)
             (argument {^ 1}) (2 ((cat student))))))
```

Figure 6-14: An FD set description

In this format for set descriptions, most features are optional (as is always the case in FDs). Thus, if only cardinality and kind are instantiated, the set is specified by quantity. If only extension is instantiated, the set is specified as a list.

The intension and reference specifications are mapped by the lexical chooser to modifiers of an NP. Both of these conceptual slots correspond to *restrictive* modifiers, since they are part of the definition of the conceptual element. Another class of modifiers, non-restrictive modifiers, are added to a nominal group to add information about the referent that is not part of its definition. A non-restrictive modifier can always be paraphrased by a conjunction of two clauses, *e.g., AI, which is difficult, has many assignments* can be paraphrased into *AI has many assignments {and} AI is difficult.* So at the conceptual level, the non-restrictive modifier is represented as a separate relation, that just happens to involve the same participants as the matrix clause.

In ADVISOR II, non-restrictive modifiers are used to merge a floating argumentative evaluation onto an existing noun group. Examples of this usage of non-restrictive modifiers are:

Analysis of algorithms, which is very theoretical.
Four assignments, which is a lot.

Besides this very specific use of non-restrictive modifiers, sophisticated text planning techniques must be used to decide to use non-restrictive modifiers in NPs. A most promising approach to this type of planning is proposed in [Robin 92b]. Robin's approach is based on a revision model, where a first draft of the text is built, with simple NPs, and additional information is hooked up where the syntax allows it, for example, in the form of non-restrictive modifiers.

To represent non-restrictive modification at the conceptual level, I allow adding a slot `modifier` to any set or individual description. The modifier slot contains the description of a relation, and the argument of the relation which corresponds to the entity being modified is indicated by the `argument` feature. An example of such input is shown in Fig.6-15.

Non-restrictive modifiers when applied to a set have two possible semantic interpretations, distributive and collective:

```
(def-test non-restrictive
  "Analysis of algorithms, which is very theoretical."
  ((cat class)
   (name analysis-alg)
   (modifier ((cat class-relation)
              (name area)
              (argument {^ 1})
              (2 ((cat field) (name theory)))))))
```

Figure 6-15: Input for a non-restrictive modifier

Four assignments, which are difficult.
Four assignments, which is a lot.

In the first NP the relative clause denotes a property of each element of the set (distributive meaning), in the second, the relative denotes a property of the set as a whole (collective meaning). By default, a modifier is interpreted as a distributive modifier. To account for collective modifiers, I have defined a class of conceptual input known as `collective-set`. An example of collective-set is shown in Fig.6-16.

```
(def-test collective-set
  "Four assignments, which is a lot."
  ((cat collective-set)
   (set ((kind ((cat assignment)))
         (cardinality 4)))
   (modifier ((cat evaluation)
              (scale cardinality)
              (argument {^ evaluated})
              (evaluated {^2 set})
              (value +)))))
```

Figure 6-16: Input for a collective modifier

In summary, set descriptions contain up to six features: extension, cardinality, kind, intension, reference and modifier. Modifier contains a non-restrictive relation. The other five features define the set. This set specification format is justified by several requirements on the lexicalization of NPs.

Specifically, because I want the lexical chooser to be able to produce judgment quantifiers such as *many* and *few*, I have identified the need to use intensional features and to distinguish between reference and intension modifiers in set descriptions. The semantic properties of judgment quantifiers which require such features in the input are presented below in Sect.6.6.

Because I want to be able to generate all sorts of nominal modifiers, classifiers, describers and qualifiers, I represent uniformly all modifiers as binary relations. The conceptual representation is not committed to a specific type of linguistic modifiers as is often the case when distinguishing between predicative modifiers and attributes at the conceptual level (as is done for example in [Dale 88]).

Finally, the conceptual representation allows the merging of argumentative evaluations into NPs using non-restrictive modifiers.

6.5.2. Summary of the Syntax of the NP: Output

The grammar for NPs was presented in Sect.5.3. The NP group contains the four types of modifiers that have been distinguished in the syntactic description of the NP in Sect.5.3, in addition to the determiner sequence:

- Determiner Sequence: *many of the same topics.*

- Describer: *an interesting topic.*
- Classifier: *a programming class.*
- PP Qualifier: *a class in AI*
- Clause Qualifier: *a class which AI covers.*

The determiner sequence is determined by using a set of features placed at the top level of the NP. The lexicalization of the determiner sequence is described in Sect.6.6. I focus in this section on how the NP modifiers of the four different types are derived from a conceptual input of the same type - a binary conceptual relation. I also identify which sites in the NP structure can serve to express an argumentative evaluation.

6.5.3. NP Planning 1: High-level Structure

The previous two subsections have defined the representation of sets and other conceptual elements which can be realized by NPs on one hand, and the syntactic representation of NPs expected by SURGE on the other hand. I now present the process mapping the conceptual input to this set of syntactic features. I refer to this process as *NP planning*. I distinguish between two stages in the NP planning:

- First, the high level structure of the NP is determined. This includes selecting a head for the NP, determining which conceptual relations will serve as modifiers, and deciding whether an apposition will be used.
- Second, each conceptual modifier is mapped to one of the four types of syntactic modifiers.

In this section, the high level structure planning of the NP is presented. I first enumerate different structural patterns which an NP can have and identify a set of four pragmatic features controlling the selection of a pattern. Finally, I explain how the features controlling the realization of the determiner sequence are derived from the conceptual representation.

To simplify the discussion in this subsection, I only consider how sets with their modifiers are mapped onto an NP structure. Individuals can be either considered as sets with one element or easily mapped to a proper noun phrase.

As presented above, the representation of sets contains up to five major features:

- **The extension list**: a list of all the elements in the set.
- **The intension predicate**: a binary relation defining a property true of all the elements of the set.
- **The kind**: the semantic category of all the elements.
- **The reference set**: a set defining the domain out of which the intension predicate selects the elements of the set.
- **The cardinality**: the number of elements in the set.

Not all these features are always instantiated in a set description. For example, it is possible to define a set only by its cardinality, or only by specifying the extension list.

Each of the elements of the set definition, when it is present, can be realized in different ways. For example, the extension can be realized by a conjunction listing all the elements one by one. The cardinality can be expressed by an exact value, as in *six assignments*, by an evaluation, as in *many assignments*, or not at all, as in *assignments*.

When several of these features are realized together, the structure of the NP becomes complex. For example, if both extension and intension need to be realized together, an apposition must be selected, as in *interesting topics, Vision, NLP and Knowledge Representation.*

To enumerate the different NP patterns which can be generated from this description, I define four realization features, which control each how an aspect of the set representation is realized. The features with their possible values are:

- **realize-extension**: yes/no
- **realize-intension**: yes/no/clause
- **realize-reference**: yes/no/clause
- **realize-quantity**: evaluation/cardinal/no

The value of these features determine which element of the set description must be realized in the NP, and how. A value of `no` indicates that the element should not be realized. A value of `yes` indicates that it should be, in any possible way. The following values have a special meaning:

- Both intension and reference can be realized by the position of the NP in a matrix clause. For example, consider the set of topics being covered in AI. The property ''to be covered in AI'' is the intension of the set description. If this set is realized in the context of the clause *AI covers many topics*, then the intension is realized by the fact that the NP appears in object position of the clause, not by any of the elements of the NP itself. In this case, I indicate that the realization is of mode `clause`.
- The quantity (which is a more general term than just cardinality for non countable sets), can be realized in two modes: cardinal, that is, by stating the precise number of elements in the set, or evaluation, that is, by evaluating the number of elements according to some criterion.

These four features can be combined into 2*x*3*x*3*x*3=54 ways. Of these 54 ways, however, certain combinations are invalid, because they do not express any information at all about the set, or because they require the cooccurence of two incompatible features. To illustrate how the enumeration works, consider the set of topics defined in extension by the elements *Vision*, *NLP* and *Robotics*, and in intension by the property that the topics are interesting. The reference set is the set of topics covered in AI. The cardinality is 3 and the cardinality of the reference set is 10. Figure 6-16 enumerates the possible NP patterns, corresponding to each combination of the four realization features. Only 18=2*x*3*x*3 combinations are listed, ignoring the subdivision introduced by the `realize-quantity` feature. Of these 18 classes, only 9 are valid.

The notation in Fig.6-16 is **Int:y Ext:y Ref:n** to indicate that the features `realize-Intension`, `realize-Extension` and `realize-Reference` have for value `Yes`, `No` or `Clause`. *V, N & R* stands for *Vision, NLP and Robotics.* The first line of the table illustrates the effect of the `realize-quantity` feature. Since the choice of the realization mode of the quantity has little effect on the overall structure of the NP, I have not listed it in the other lines. One constraint on the realization of quantity which does not appear in the table is that, if the quantity must be realized and the only other element realized is the extension, then a head must be introduced anyway just to carry the determiner, for example, as in, *3 topics, V, N & R*. This implies that if `realize-quantity` is not `no`, then `kind` must be instantiated. If it is not, then a generic noun, *thing* or *stuff*, is used as in *2 things, mathematics and essays.*

Of the 18 patterns listed in Fig.6-16, only 9 are valid realizations of the set description. Two combinations of features are impossible to satisfy because they require the NP to be a participant in two clauses simultaneously (the two combinations containing Int:c and Ref:c). Seven combinations are incomplete, because they do not express enough information to identify the set. For example, the combination Int:y Ext:N Ref:N (INC1) yields *many interesting topics* but does not specify that the topics are those covered in AI.[54] Note that when the intension slot of the set description is not instantiated, the set of patterns is different from the one shown in the table.

When all elements are realized, the overall pattern for the realization of the set description is `(kind mod1 mod2) conj`, that is, up to two modifiers attached to a head noun realizing the kind, in apposition to a conjunction of nouns realizing each individual element of the extension. The two modifiers can be realized as two describers, one describer and one classifier, one describer and one qualifier, two classifiers, one classifier and one qualifier, or two qualifiers. The lexical chooser, therefore, must select how to map each semantic element, like reference and intension, to one of these syntactic elements, and for each modifier determine the syntactic category of the modifier - adjective, noun, PP or relative clause. The apposition can be realized in different ways:

- As a simple apposition, *e.g.*, *two topics, Vision and Robotics*.

[54]When judging the completeness of a realization, I assume that no information on the set can be inferred from the discursive context. Basically, this means that I consider only first references to the set in discourse.

Int:y Ext:y Ref:y	*John wants to learn about many interesting topics covered in AI - V, N & R.* *John wants to learn about three interesting topics covered in AI - V, N & R.* *John wants to learn about interesting topics covered in AI - V, N & R.*
Int:y Ext:y Ref:n	*John wants to learn about many interesting topics - V, N & R.*
Int:y Ext:y Ref:c	*AI covers many interesting topics - V, N & R.*
Int:y Ext:n Ref:y	*John wants to learn about many interesting topics covered in AI.*
Int:y Ext:n Ref:n	*John wants to learn about many interesting topics.* [INC1]
Int:y Ext:n Ref:c	*AI covers many interesting topics.*
Int:n Ext:y Ref:y	*John wants to learn about many topics covered in AI - V, N & R.*
Int:n Ext:y Ref:n	*John wants to learn about V, N & R.*
Int:n Ext:y Ref:c	*AI covers V, N & R.* [INC2]
Int:n Ext:n Ref:y	*John wants to learn about many topics covered in AI.* [INC3]
Int:n Ext:n Ref:n	*John wants to learn about many topics.* [INC4]
Int:n Ext:n Ref:c	*AI covers many topics.* [INC5]
Int:c Ext:y Ref:y	*Many topics covered in AI - V,N,R - are interesting.*
Int:c Ext:y Ref:n	*V, N & R are all interesting.* [INC6]
Int:c Ext:y Ref:c	impossible [2 clauses]
Int:c Ext:n Ref:y	*Many topics covered in AI are interesting.*
Int:c Ext:n Ref:n	*Many topics are interesting.* [INC7]
Int:c Ext:n Ref:c	impossible [2 clauses]

Figure 6-17: List of NP patterns

- As a parenthetical, *e.g., two topics (Vision and Robotics)*.
- With a ‘‘connective’’, *e.g., two topics including Vision and Robotics, an interesting topic, such as NLP*.

The choice between these options is currently made randomly, providing variety in the output.

In addition, the order of the two appositive conjuncts can be switched:

- Intension first: *a very theoretical topic, Logic.*
- Extension first: *Logic, a very theoretical topic*.

The second option is selected when either realize-intension or realize-reference is set to yes (because otherwise an NP like *Logic, a topic* would be generated), realize-quantity is not set to evaluation (because this would lead to generating * *Logic and Calculus, many theoretical topics*) and, of course, realize-extension is set to yes. When these conditions are met, if the cardinality of the set is one, the second form is always selected, otherwise, the choice between the first and second form is made randomly.

Having enumerated the different structural patterns for NPs, and identified four realization features controlling their selection, I now focus on the definition of pragmatic features which can determine the value of these four realization features. I have found the following factors to be of importance:

- **Information available**: if, for example, the list of all elements in the set is not known, or the cardinality is not known, then the corresponding realization feature is set to `no`. If it is available it does not necessarily need to be realized.
- **Argumentative function**: when an argumentative evaluation exists on the cardinality of a set, `realize-quantity` is set to `evaluation`.
- **Style and length of expression**: the extension when realized by a conjunction yields a long expression for the NP. If conciseness is required the `realize-extension` is set to `no`.

- **Pragmatic function**: listing the elements of a set and describing the set in intension do not fulfill the same communicative goals. The difference is related to the naming/describing distinction studied for example in [Downing 77] (cf. p.184 for a discussion). It is also related to the definiteness of the NP. I have not looked at how these issues can determine the realization features discussed here.

The output of the first stage of the NP planning process is a generic linguistic structure of the form `(apposition (head mod1 mod2) list)`. This structure plays a role similar to the generic `roles` feature used at the clause level presented in p.170. The ability to map a conceptual input onto this level of NP structure is a major source of paraphrasing power. As discussed in this section, for a fully specified set, 27 valid patterns have been identified to realize the set structure; in addition the choice of apposition style multiplies this number by six. Finally, each one of the two modifiers can be realized in up to four ways, as discussed now. This level of variety (4x4x6x27) illustrates the power of a compositional approach to text generation as opposed to phrasal lexica or templates. In addition, and this is the crucial claim in this thesis, the selection between the many realizations of a set as an NP is sensitive to the pragmatic situation, making the whole generator more fluent.

6.5.4. NP Planning 2: Choice of Syntactic Modifier Type

Once the overall structure of the NP is determined, the lexical chooser determines the category and position of the modifiers in the NP. For each modifier, the choice is between describer, classifier, PP qualifier and relative qualifier. The strategy I follow is to map in priority a modifier to a classifier if possible, else to a describer, and in last resort to a qualifier. These priorities can be changed if an argumentative evaluation must be realized on the current NP. Figure 6-18 summarizes the approach followed for the stage of NP planning.

I first determine when each type of modifier can be used. In the following discussion, the modifier relation is represented by the expression `x R y`. One of the arguments, `x` or `y` is the head of the NP. I call the second argument the *external argument*, following [Williams 80] as used in [Levin and Rappaport 86].

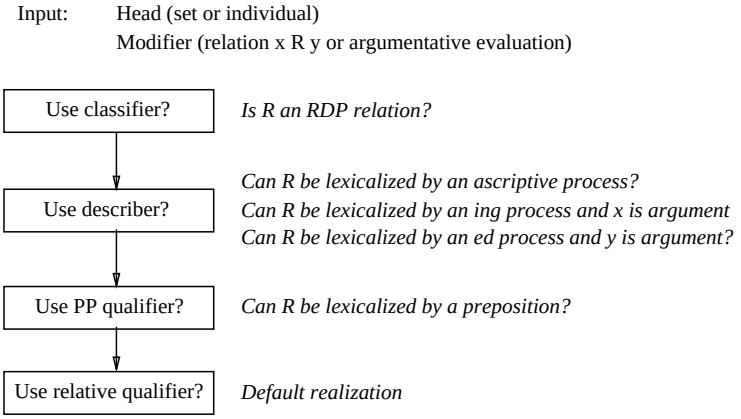


Figure 6-18: Mapping of Conceptual to Syntactic Modifiers

6.5.4.1. When Can a Classifier be Generated?

Defining the semantics of the modification in the case of classifying modifiers is a challenging task; yet, [Levi 78] provides an excellent study of the general phenomenon, which fits very well within the lexicalization process of a text generator. Levi defines broadly complex nominals as the three classes of NPs with pre-modifiers of the form:

- Noun-noun modifiers: *apple cake*, *autumn rains*.
- Nominalizations: *Markovian solution*, *film producer*, *quantifier lowering*.
- Non-predicative adjectives: *electric shock*, *musical clock*, *musical criticism*.

All these cases correspond to a pre-modification where a transformation *X N* into *N is X* is not semantically valid. Another characteristic of these modifiers is that:

> their meaning appears to change depending on the head noun that they modify. For example, I understand *musical clock* to mean 'a clock that produces music'; yet I also know that *musical criticism* could hardly be construed as 'criticism that produces music', but must mean instead 'criticism of music', where music functions as the object of a nominalized verb. [Levi 78, pp.3-4]

In this work, I call all three types of modifiers, classifiers, to remain consistent with the syntactic analysis provided in Sect.5.3. Because the meaning of a classifier modification can be so diverse, there is a tendency to view classifier-head groups as frozen collocations in the domain, that are idiomatic and need to be stored as a group in the lexicon. This is the approach pursued, for example, in [Smadja 91a] for text generation. There are, however, as demonstrated by Levi's analysis, **productive** processes that generate classifier modification. Levi's thesis advances the following points (cf. p.50):

- Classifier modifications are derived from an underlying NP containing a head noun and a full sentence.
- Classifier modifiers are inherently and regularly ambiguous over a predictable and relatively limited set of possible readings. The potential ambiguity is reduced in discourse by both semantic and pragmatic considerations.
- Complex nominals are all derived by just one of two syntactic processes: the **deletion** or the **nominalization** of the predicate in the underlying sentence. In this work, I focus on predicate deletion only. For the classifiers resulting from such deletion, a small set of **Recoverably deletable predicates** (RDPs) can be identified, such that ''only its members and no other predicates can be deleted in the formation of complex nominals; the members of this set are: CAUSE, HAVE, MAKE, BE, USE, FOR, IN, ABOUT and FROM.''

The claim is, therefore, that classifiers can be produced systematically from a form similar to post-modifiers by using the operation of predicate deletion when it is applicable. Levi advances two motivations explaining why classifiers are preferred over post-modifiers: they make up for more compact expression (*e.g.*, compare *an AI topic* with *a topic (which is) about AI*); and they can be used as ''names'' for objects, as opposed to descriptions. This corresponds to the communicative need ''to refer to an entity which possesses no name of sufficient specificity'' [Downing 77]. Compare, for example, the following references [Levi 78 p.61]:

Hey, there's a cloud that looks like a kangaroo.
Hey, look at that kangaroo cloud.

In the second form, the cloud is named and not described, and the description is used to fulfill a classification function. These two motivations explain why there is an incentive to produce new classifiers that cannot be expected to be found in a lexicon.

RDP	Classifier subject	Classifier object
CAUSE	mortal blow	viral infection
HAVE	picture book, apple cake	government land, feminine intuition
MAKE	musical clock	consonantal patterns
USE	manual labor, steam iron	-
BE	mammalian vertebrate, soldier ant	-
IN	field mouse, marine life	-
FOR	horse doctor, arms budget	-
FROM	olive oil, rural visitors	-
ABOUT	tax law, linguistic lecture	-

Figure 6-19: Levi's Recoverably Deletable Predicates. From [Levi 78, p.76]

Levi's claim is that a closed set of 9 generic predicates can explain the production of a large class of classifier modifiers. Figure 6-18 lists Levi's recoverably deletable predicates, with examples of each relation.

In the ADVISOR II domain, the following are examples of classifier modifiers, with their classification in terms of RDPs:

an AI topic	IN
an Intro to Programming assignment	IN
an essay assignment	BE
a theory course	ABOUT
a quiz grader	FOR

The lexical chooser classifies each domain-specific conceptual relation as a specialization of one of Levi's 9 generic RDPs. In FUF this is accomplished by defining a type hierarchy where domain relations are defined as instances of the 9 Levi's roots:

```
(define-feature-type in-pred (topics assignments))
(define-feature-type about-pred (area))
(define-feature-type be-pred (hw-type))
```

When the classifier is a non-predicative adjective, the deletion of the RDP is also accompanied by a morphological transformation of the argument of the deleted predicate. For example, *musical box* is related to a form involving *music*, *i.e.*, *a box that produces music*. Levi provides a sequence of seven arguments supporting that view, for example, that most non-predicative adjectives are not degree adjectives (they cannot be modified by *very* [Levi 78, Chap.2]). So the steps followed to produce a classifier from a conceptual representation where an object is modified by a relation are:

1. Determine whether the modifying relation is an instance of an RDP.
2. Form a classifier of the appropriate part of speech with the second argument of the relation.

The lexical chooser determines if a conceptual relation can be realized by a classifier by checking its lexicon entry. If the relation is an instance of one of Levi's 9 Recoverably Deletable Predicates, then a classifier modifier is produced. The external argument is mapped to the classifier. The classifier must then be lexicalized. Two options are then possible: the classifier is mapped to a non-predicative adjective or to a noun, entering into a noun-noun modification. The choice between these two options may depend on the relation being deleted, on the head of the NP or just on the external argument. If the choice depends on both the relation and the head of the NP, then the expression is probably a frozen collocation and is entered as a single chunk in the lexicon. It is not derived through the mechanism of RDPs. In the ADVISOR II domain, the choice of the part-of-speech of the classifier seems to depend only on the external argument. So the classifier is simply lexicalized with the feature `non-predicative` set to `yes` to force the selection of the appropriate form for the external argument. For example, `((cat field) (name theory))` is lexicalized into the noun *theory* (the default), but `((cat field) (name theory) (non-predicative yes))` is lexicalized into the adjective *theoretical*.

This application of Levi's theory is the first technique ever used to produce non-predicative modifiers in a compositional manner in text generation. This problem was listed as an open problem by Reiter in his study of NP generation [Reiter 90]. There are some problems in applying this technique that deserve further study: first, Levi's predicates are vague and not defined very formally; second, there remains to establish clear criteria to determine when an instance of RDP is to be realized by a classifier as opposed to another form. I discuss in Sect.6.5.4 how the argumentative properties of lexical items can force the realization of an RDP as a qualifier. Other factors, including the factors of brevity and no-unnecessary components used by Reiter, should also play a role in determining which modifier is to realize a RDP.

6.5.4.2. When Can a Describer be Generated?

To determine if a describer can be generated, the following conditions are checked: either the relation can be lexicalized by an ascriptive clause, or the relation can be lexicalized into a clause whose main verb can be transformed into an *-ing* or *-ed* premodifier, with the appropriate mapping between the head and external argument and the oblique 1 and 2 arguments of the clause.

The derivation of *-ed* premodifiers is studied in depth in [Levin and Rappaport 86]. *-ed* premodifiers are known as ‘‘objective modifiers’’ in traditional grammars, because the head appears to correspond to the object of the clause: *a well taught class* corresponds to *someone teaches the class well*, where *class* is the ‘‘object’’ of *teach*. The problem is that not all pairs *verb-object* can give rise to a form *verb-ed object* where *verb-ed* is an adjective. For example, * *a helped man* or * *a thanked man* are not valid modifiers. An alternative analysis based on the thematic role of the head in the underlying clause looks, therefore, more appealing: an *-ed* adjective is usable if the head of the NP corresponds to the affected (or theme) role. Levin and Rappaport argue against the thematic condition based on an analysis of dative verbs, for example, *to feed* as in *feed some cereal to the baby* which can produce *unfed baby* but not *unfed cereal*, even though *cereal* fills the theme (or affected) role. Another class of verbs which do not fit with the thematic condition is the *spray/load* family, as illustrated by:

load the truck; the recently loaded truck.
load the hay; recently loaded hay.

(from [Levin and Rappaport 86, p.635])

where both the affected and the location argument can become the head of the NP. For certain verbs, only the location argument can become the head (*e.g.*, *stuff*, *cram*). L&R, therefore, propose instead an account which seems to rely mainly on the syntactic constraint that the direct object can become head with an *-ed* modifier.

The second issue to determine whether both *-ed* and *-ing* modifiers are appropriate is to check whether the external argument can be omitted. For example, from the relation *a course interests the hearer* can the NP *an interesting course* be derived, even though there is no mention of *the hearer* in the NP? The alternative is *a course interesting for you* in the post-modifier position. This alternative is discussed further in Sect.6.7.

To simplify the implementation, I require each verb entry in the lexicon that can be used as a describer to specify it using the features `(ing-adj yes)` or `(ed-adj yes)`. This feature indicates that the *-ing* or *-ed* form of the verb can be used as an adjective and that the external argument does not need to be realized. To check whether the head matches the appropriate argument of the verb in the underlying clause I use the subcat frame of the verb. For example, the lexical entry for the verb *to interest* is shown in Fig.6-20. Using the subcat frame to check the argument compatibility is similar to using the thematic condition criticized by L&R. But as they themselves point out, the thematic condition is only challenged by the ‘‘marginal data from the exceptional dative verbs and from the relatively small class of *spray/load* verbs’’ [Levin and Rappaport 86, p.657]. So the simplification seems warranted by the generality of its application. Note also that the ‘‘worse that can happen’’ is that a relative clause will get generated when an *-ed* describer could have been used, so the cost of the simplification is really negligible.

```
((name interest)
 (lex-cset ((+ {^3 roles 1 semr} {^3 roles 2 semr})))
 ({^} ((cat verb-group)
       (type lexical)
       (lex "interest")
       (ing-adj yes)
       ({^ lex-roles} {^ roles})
       (subcat ((1 {^3 roles 2})
                (2 {^3 roles 1}))))))))))
```

Figure 6-20: Lexical entry for ‘‘to interest’’

Finally, a describer can be realized to express an argumentative evaluation when the following conditions are met:

- The evaluated element of the evaluation is coindexed with the head of the NP.
- The scale and orientation of the evaluation can be lexicalized by a scalar adjective.

The representation of the adjectives and the lexicalization of the scales is presented in Sect.6.7.

6.5.4.3. When Can a PP Qualifier be Generated?

The last option to realize an NP modifier is to use a qualifier. Qualifiers can be PPs or relative clauses. The lexical chooser tries to generate a PP over a relative clause when possible because it is more compact. The decision between PP and relative is based on the lexical entry of the conceptual relation in the modifier. A relation is mapped in general to a verb. Some relations can also be mapped to a preposition. For example, the relation *topics* which holds between a course and a topic can be realized by the preposition *about*, or by the verb *cover*. So if the *topics* relation is to be realized as a qualifier, I choose the realization *a class about AI* over *a class which covers AI*. The lexical entry for this relation is shown in Fig.6-21.

```
((name topics)
 (alt (
       ;; Realize as a PP: 1 about 2
       (({^} ((cat pp) (prep ((lex "about")))
                       (np ((semr {^3 semr roles 2 semr})))))
        (lex-cset ((+ {^2 np semr}))))

       ;; Realize as a verb: 1 covers 2, 1 deals with 2...
       (({^} ((cat verb-group)
              (type lexical)
              ({^ roles} {^ lex-roles})
              (subcat ((1 {^3 roles 1}) (2 {^3 roles 2})))
              (lex "cover")))
        (lex-cset ((+ {^3 roles 1 semr} {^3 roles 2 semr})))))))
```

Figure 6-21: Lexical entry for the topics relation

6.5.4.4. Interaction NP Planning with Argumentation

To realize a modifier, the lexical chooser tries the options in the following order: first classifier, then describer, PP qualifier and finally relative modifier, thus giving priority to the most compact realization. This preference can be overridden, however, if the words inserted by the classifier or describer modification introduce unwanted argumentative connotations. For example, consider the realization of the modifier `area(class, theory)`. The lexical entry for `area` indicates that it is an RDP, and, therefore, a pre-modifier can be used. The pre-modifier lexicalization of `theory` is the adjective *theoretical*. But this adjective is also a degree adjective which realizes a positive argumentative evaluation of its head on the scale of `theory`. If this evaluation is not part of the input, then the adjective is not selected, and instead a qualifier is selected: *a course about theory* is selected over *a theoretical course*.

Non-restrictive modifiers are only added to a NP to convey argumentative evaluation when a floating evaluation needs to be realized. Of the different ways of expressing an argumentative evaluation, the non-restrictive modifier is the least preferred, because it is the least compact. There is one case where the non-restrictive modifier is often used: when the cardinality of a set is evaluated and the `realize-quantity` is set to cardinal. That is, the determiner must express the exact quantity and an argumentative judgment must also be expressed. In this case, a collective modifier is produced, as in *6 assignments, which is a lot*. The non-restrictive modifier is used because the ‘‘natural’’ site of expression of the argumentative evaluation, the determiner, is already occupied by the cardinal. The mechanisms used to produce the non-restrictive modifier are described in Sect.6.7.3.

6.5.4.5. Summary: Mapping Conceptual Modifiers to Syntactic NP Modifiers

In summary, the second stage of the NP planning process determines how conceptual modifiers of an NP are mapped onto syntactic NP modifiers of four different types: classifiers, describers, PP qualifiers and relative qualifiers. These four types are tried in this order, giving preference to the most compact forms when they can be used. The following criteria are used to determine when each type of syntactic modifier can be used when the input modifier is a conceptual relation `xRy`:

- A classifier can be used when the relation `R` is an instance of one the nine recoverably deletable predicates identified in [Levi 78].

- A describer can be used:
  - When the relation `R` can be lexicalized by an ascriptive relation
  - When `R` can be realized by a process which carries either the `ing-adj` or `ed-adj` feature, the head argument corresponds to the appropriate thematic role of the relation and the external argument does not need to be expressed.
  - When an argumentative evaluation must be realized, the head of the NP realizes the evaluated element of the evaluation and the scale and orientation of the evaluation can be realized by a scalar adjective.
- A PP-qualifier can be used when `R` can be lexicalized by a preposition.
- In all other cases, a relative-qualifier is generated.

This mapping procedure justifies the fact that in the conceptual representation, all modifiers are represented uniformly as relations. This is another example of the principle that the less the conceptual representation is committed to a linguistic perspective, the more paraphrasing power the lexical chooser can implement.

6.5.5. Conclusion: Constraints on Lexicalization at the NP Level

I have presented in this section the construction of NPs in ADVISOR II and their lexicalization. I have focused on the lexicalization of sets of countable entities. In summary, the lexical decisions made when constructing an NP for such a set are:

- NP Planning 1: Build a generic high-level structure for the NP of the form `(apposition (head mod1 mod2) (conj))`.
- Map each modifier to one of the syntactic function of classifier, describer, PP qualifier or relative qualifier.
- Lexicalize the head and each modifier, taking its syntactic position into account.
- Set the determiner features (this is discussed in the next section).

These decisions rely on the following information in the conceptual input and in the lexicon:

- Intension and reference are distinguished in the input by the evaluation functions, to reflect an argumentative perspective on the set.
- All modifiers are represented as binary relations in which one argument is filled by the set. The realization of a relation as classifier, describer or qualifier is made depending on its conceptual type.
- Argumentative evaluations of the set can be realized by forcing the selection of lexical items with argumentative connotations among the modifiers and/or the determiner (forcing the choice of words like *hard* instead of *difficult*, or of *many* instead of *some*), or by adding a non-restrictive modifier (*e.g.*, *four assignments, which is a lot.*)
- The lexicon contains entries indexed by conceptual classes for the nouns. Relations that are instances of one of 9 Recoverably Deletable Predicates are marked as such and mapped to classifiers. In such a case, the external argument of the modifier relation is lexicalized as a non-predicative modifier, which can be a noun or an adjective. Relations that can be realized by an ascriptive process type are mapped to describers or PP qualifiers. Other relations are realized as relative qualifiers.

The main contributions of this lexicalization technique at the NP level are:

- The definition of the task of NP planning. NP planning is required when mapping a purely conceptual input onto a complex NP structure. The NP planning process introduces a major source of paraphrasing power in the generator, and most important, it is sensitive to pragmatic features: argumentative orientation, style, information available to the hearer.
- The generation of non-predicative modifiers in a compositional way, based on systematic principles.

6.6. Generation of the Determiner Sequence

This section presents the techniques used to set the features controlling the generation of the determiner sequence in an NP. The case of the determiner sequence is different from the clause and NP levels for two reasons:

- The determiner sequence is a closed-system: output of the lexical chooser does not consist in lexical items but in a set of features which SURGE uses to select one of a closed set of determiners.
- There is no planning stage in the lexicalization of the determiner sequence, in contrast to the clause and NP levels.

The determiner sequence is located within the NP, and following [Barwise and Cooper 81], I view NPs as a whole as the expression of *generalized quantifiers*, as opposed to simply the determiner sequence. The input to the determiner lexicalization process, therefore, is a set specification, as described in Sect.6.5.1. The output, similarly, is a set of features appearing at the NP level.

In this section, I first present the part of the input specification which is used by the determiner lexicalization process. I then list the features of the NP syntactic description which are set by this lexicalization and determine the generation of the determiner sequence. I then proceed to the description of the mapping from input to output. The determiner sequence is, among the syntactic constituents, one of the most sensitive to the argumentative orientation of an utterance. I therefore describe in Sect.6.6.1 some properties of the class of determiners called *judgment determiners* which express argumentative evaluations. Judgment determiners include *many*, *few*, *a lot* etc. I derive from this analysis the need to distinguish between semantic and pragmatic validity in generation; some constraints on the usage of determiners and the requirement of an additional feature in the input, called `typical-size`, to appropriately select judgment determiners. Next, I present the semantic property of *monotonicity*, introduced in [Barwise and Cooper 81], and explain how it is related to the notion of argumentative orientation, defining a linguistic test allowing to categorize determiners as either positive or negative. Finally, I present a semantic property of determiners called the *intersection condition*, and explain how the determination of the determiner sequence can influence NP planning. This property justifies the distinction in the conceptual input specification of sets between `intension` and `reference` slots. Finally, I summarize how each feature in the output is set, based on the argumentative evaluation to be realized in the determiner sequence.

6.6.1. Input: Sets and Argumentative Features

The input to the determiner selection procedure is a set description, annotated by an optional argumentative evaluation. The format of the set specification was presented in Sect.6.5.1. It includes five main features: kind, cardinality, intension, extension and reference.

The argumentative evaluations which affect the selection of the determiner sequence are all on the scale of cardinality.[55] An argumentative evaluation of the cardinality of a set consists of a judgment of the set as either large or small. An example of input specification is shown in Fig.6-22.

6.6.2. Output: Relevant Features

The features controlling the generation of the determiner sequence have been described in Sect.5.4. The complete set used by SURGE includes 24 features. Not all these features are supported by the lexical chooser. This is a case where the syntactic realization component can provide much more variety than the lexical chooser can account for.

A first group of 9 features are derived in a simple manner:

---

[55]Recall that I only consider in this work countable sets of discrete elements. Therefore cardinality is used here instead of the more general scale of quantity.

```
((cat set)
 (kind ((cat topic)))
 (cardinality 7)
 (reference ((cat set)
             (kind ((cat topic)))
             (cardinality 10)
             (intension ((cat class-relation) (name area)
                         (argument {^ 1})
                         (2 ((cat field) (name AI)))))))
 (intension ((cat user-relation) (name interest)
             (argument {^ 1}) (2 ((cat student)))))
 (ao ((cat evaluation)
      (evaluated {^ argument})
      (scale ((name cardinality)))
      (value +))))
```

Figure 6-22: Input for the lexicalization of the determiner sequence

- **Definite:** I have avoided in this work the issue of determining the definiteness of NPs. A principled approach to this intricate issue has many implications on the design of a generation system. Dale provides an excellent analysis of the issues in a complete generation system [Dale 88, Sect.2.3, 5.1 and 5.5]. The ADVISOR II implementation relies on drastic simplifications: individuals are assumed to be definite, sets are indefinite.
- **Countable**: lexical feature of the head noun.
- **Number**: based on the cardinality.
- **Reference-number**: based on the cardinality of the reference set.
- **Exact**: set to yes if `(realize-quantity cardinal)` is set. Otherwise, set to no.
- **Case**: set by the syntactic realization grammar, depending on the position of the NP in a clause.
- **Head-cat**: inherited from the head (pronoun, common, proper) in the syntactic realization grammar.
- **denotation-class**: The categories quantity, season, institution, transportation, meal, illness are defined in the ontology of the CLASSIC knowledge-base, and all concepts that are of these categories must be defined as specializations of these concepts in CLASSIC. When this is done, recall that a FUF type hierarchy is derived from the CLASSIC concept hierarchy, and the cat of the semantic representation then unifies with the symbols quantity, season, etc.
- **Cardinal**: the cardinality when realize-quantity is cardinal, else none.

The following 10 features are not considered at all by the current lexical chooser, and their value is hard-wired:

- **Fraction**: Not used in this domain. Always set to none.
- **Evaluative and Evaluation**: Not studied. Always set to no. (Note that evaluative determines the use of the determiners *too many*, *too much*, *enough*), it does not correspond to the argumentative evaluations studied in this thesis.
- **Interrogative**: always set to no.
- **Possessive**: always set to no.
- **Distance**: always set to none. Distance determines the use of demonstrative determiners (*this* or *that*).
- **Ordinal**: always set to none. Ordinal determines the use of *first*, *second* etc and of *last* and *next*.
- **Status**: always set to none. Status determines the use of a deictic adjective such as *given*, *above*, *mentioned* etc.
- **Selective**: is set to Nil. Its value is constrained by the other features. Selective forces the selection of ‘‘partitive’’ determiners, such as *one* or *either*, or *some of* etc.

- **Partitive**: is set to Nil. Its value is constrained by the other features.

The following 5 features are related to the argumentative evaluation which must be realized at the NP level. The determiner is the natural site of realization for all argumentative evaluations on the cardinality scale, and the following features establish the connection between the argumentative evaluation and the determiner sequence:

- **Total**: when argumentative orientation is positive and the cardinality of the set is known and equal to the cardinality of the reference set, the total is set to +. If argumentative orientation negative and the cardinality is 0, the total - is set. Otherwise, total is set to none.
- **Orientation**: is set to the value of the argumentative orientation (+ or -). If there is no argumentative evaluation, it is set to none.
- **Comparative**: Set to yes when the argumentative evaluation is of type relative. No otherwise.
- **Superlative**: Set to yes when the reference set is given and its cardinality known, and the cardinality is larger than half of the reference set cardinality.
- **Degree**: The value of degree depends on a comparison between the cardinality and a typical size, whose determination is detailed below.

I now focus on these 5 features and provide more detail on how their value is determined by the lexical chooser. I first present some semantic properties of the class of judgment determiners introduced in linguistics. I then derive requirements on the way these features can be set.

6.6.3. Argumentation and Determiners: Judgment Determiners and the Setting of Degree

The determination of the degree is a delicate issue. The degree feature determines how to select among *a few*, *some*, *many* or *a (large, great, incredible...) number*, if the orientation is +, and among *few*, *a (small, tiny...) number* if the orientation is -. If degree is set to none, then the determiner selected is *several* or *some*. For a given set description, with both cardinality and reference set cardinality specified, many different feature configurations can be selected. To understand which component of the generator is responsible for setting the value of these features, I need to distinguish between semantic validity and pragmatic validity. A determiner is semantically valid if it does not misrepresent the cardinality of the set. A determiner is pragmatically valid if it does not mislead the hearer about the quantity of the set and it expresses the intent of the speaker. A semantically valid determiner can be pragmatically invalid. I leave open the question as to whether a pragmatically valid determiner can be semantically invalid. For example, if the hearer intends to emphasize the difficulty of a class and says *many mathematical assignments* when the class has only one mathematical assignment, can this use of *many* with a plural noun be termed misleading? It is semantically invalid, since it implies that more than one mathematical assignment exist. But it expresses the intention of the speaker. I take in this work the conservative position that the generator should not produce this type of semantically invalid expressions.

To establish the semantic validity of a determiner, I must establish truth-conditions on its usage. There is, however, no clear truth-conditional definition for the vague determiners I consider here, *i.e.*, those that do not satisfy the intersection condition. For example, in the case of *many*, the question can be phrased as: can there be a truth-conditional constraint determining when *many* can be used, and when *many* is not true when applied to the size of a set. One truth condition is that *many* implies a plural number. So if the set has less than one element, *many* is not semantically valid. Is there a stronger requirement on *many*? Examples such as the following indicate that the meaning of *many* is not determined by the cardinality of the set it quantifies, but that a comparison is also involved:

There are many people on this bike. [3]
Few people attended the ceremony. [1,000]

In these examples, sets with cardinality 3 are referred to with *many*, while for a set with cardinality 1,000, the determiner is *few*. In [Keenan and Stavi 86, 1.2], Keenan and Stavi make the argument that determiners like *many* and *few* are not *extensional*, in the following sense:

> To say that a det *d* is extensional is to say, for example, that whenever the doctors and the lawyers are the same individuals then *d doctors* and *d lawyers* have the same properties, *e.g., d doctors attended the meeting* necessarily has the same truth value as *d lawyers attended the meeting*. [Keenan and Stavi 86, p.257]

K&S move on to show that *many* is not extensional by using the following example:

> Imagine, for example, that in the past, the annual doctors meeting has been attended by tens of thousands of doctors, and only two or three lawyers. But during the course of the year, and unbeknownst to everyone, all the doctors get law degrees and all the lawyers get medical degrees (so that doctors and lawyers are now the same) and at this year meeting only 500 doctors/lawyers show up. Reasonably then (3a) is true and (3b) is false:
> (3a) Many lawyers attended the meeting this year.
> (3b) Many doctors attended the meeting this year.
> Thus, *many (few)* cannot be treated extensionally.

One might expect that an intensional treatment could provide a formal semantic definition of *many*. But it does not seem to be the case. *Many* implies a comparison to a standard, which is most of the time not provided, and must, therefore, be considered as a pragmatic variable, bound in context. This is the conclusion reached by K&S, who, although they provide a formal denotation for an exceptionally long list of determiners, exclude the so-called value judgment determiners from the scope of formal semantics:

> In general, we feel it is best to regard *(4) Many tourists visited the zoo today* as simply indeterminate in truth value. The communicative utility of such sentences does not lie in what they literally say about the world. (4) says almost nothing about how many tourists visited the zoo today over and above a commitment to at least two. Rather the utility of such sentences lies in the fact that they enable the speaker to express his *value judgment* at the number who visited, more or less **regardless of what that number is**. Thus one who asserts (4) indicates that he regards the number of visitors as *significant*, and he may have almost any random reason for that assessment.

It is precisely the function of argumentative evaluations in the model presented in this work to account for these ''random reasons'' and to explain why the number of elements in a set is significant. Therefore, the decision to set the value of degree to +, - or none is made by the same module of the system determining that an argumentative evaluation is relevant at all, *i.e.*, by the evaluation functions. I still need to state some criteria that help determine how the evaluation function can set the value of degree, or provide the information necessary to compute it. The issue of defining degree arises again to select scalar adjectives (*e.g.*, *easy*, *hard*, *difficult*). I discuss below the notion of comparison class derived from [Klein 80]. A similar notion, called *context sets* is also introduced in [Westerstahl 84]. The basic intuition is illustrated by the case of *many*: *many* can be interpreted as comparing the cardinality of the set it quantifies with a contextually determined comparison class. For example:

There are many people in this car: *more than usually in a car.*
There are many assignments in AI: *more than in a typical CS class.*

In general, setting (degree +) means that a comparison class is relevant in the current context, and that the number of elements in the quantified set is larger than in the comparison class. In the simplest cases, this information is provided by the clause in which the NP appears. For example, the lexical entry for the relation `topics` is shown in Fig.6-23. It indicates that the typical number of topics covered in a class is 4, and that a topic is typically covered in one class. When this information is available, the lexical chooser can determine the degree of the determiner for topics in *AI covers many topics*, by comparing the number of topics in AI with 4.

```
;; Simple lexical entry with typical-size specification
((name topics)
 (lex-cset ((+ {^3 roles 1 semr} {^3 roles 2 semr})))
 ({^} (;; The constituents are class and topics under roles.
       ;; Typically 1 class covers 4 topics.
       ;; Typically 1 topic is covered by 1 class.
       ({^ roles 1} ((kind ((cat class) (typical-size 1)))))
       ({^ roles 2} ((kind ((cat topic) (typical-size 4)))))
       (:& topics))))
```

Figure 6-23: Determination of typical size

In more complex cases, the relevant typical size cannot be inferred from just the clause where the NP occurs. For example, in *AI covers many interesting topics*, the comparison seems to be between the ratio of interesting vs. non-interesting topics in AI and in other courses. This is the type of complication that fuzzy semantics proposes to address (cf. [Zadeh 84, pp.382-384] or [Lakoff 75] for example). But the fuzzy semantics approach requires the definition of a numeric function in the range [0..1] for each predicate, and models the combination of predicates as operations like multiplication or division. These operations often lead to counter-intuitive or arbitrary modeling decisions.

In these cases, where an explicit intension definition is provided, I rely completely on a domain-dependent assessment of the degree feature by the evaluation function. In general, the formalization of degree is a difficult problem that I have avoided in this work. The difficulty of the issue is discussed in Chap.8 (p.241).

6.6.4. Monotonicity and the Orientation Feature

Orientation corresponds to the positive or negative meaning of certain quantifiers. For example, *few* has a negative meaning, whereas *a few* has a positive meaning, as noted for example in [Quirk *et al* 72, p.144]:

He has few ('not many') friends and little ('not much') money.
He has a few ('some') friends and a little ('some') money.

The feature orientation distinguishes between negative and positive quantifiers. Orientation is distinct from degree, as illustrated by the fact that different degrees can be expressed for the same orientation:

AI has a little programming. `orientation +, degree -`
AI has a lot of programming. `orientation +, degree +`

The notion of orientation is related to the notion of monotonicity defined formally in [Barwise and Cooper 81, pp.184-191]. The linguistic test to determine if a determiner is monotonic increasing (equivalent to an orientation +), monotonic decreasing (orientation -) or not monotonic at all is as follows: consider two verb-phrases, VP_1 and VP_2, such that the denotation of VP_1 is a subset of the denotation of VP_2, that is, in logical terms, $VP_2(x)$ implies $VP_1(x)$. Then by checking whether the following seem logically valid, one can determine if the determiners are monotonic:

If NP VP_1, *then* NP VP_2. (NP is monotonic increasing)
If NP VP_2, *then* NP VP_1. (NP is monotonic decreasing)

[Barwise and Cooper 81, p.185] gives the following example, taking VP_1 to be *entered the race early* and VP_2 to be *entered the race*:

If { some Republican / every linguist / John / most peanut farmers / many men } entered the race early, then { some Republican / every linguist / John / most peanut farmers / many men } entered the race.

All these implications are valid, while the reverse implications do not hold. Similarly, the following implications indicate that the determiners *no, few* and *neither* are monotonic decreasing:

If { no plumber / few linguists / neither Democrat } entered the race, { no plumber / few linguists / neither Democrat } entered the race early.

Finally note that the determiners *exactly two* or *at most three* are not monotonic at all, since there is no implicative relation between *exactly three men entered the race early* and *exactly three men entered the race*.

Monotonic determiners contain the class of determiners that can be used to express an argumentative evaluation of the size of a set. Monotonic increasing determiners express a positive evaluation, while decreasing determiners express a negative evaluation. I use this property when mapping argumentative evaluations onto NPs.

The distinction between monotonic increasing, monotonic decreasing and non-monotonic corresponds precisely to the values of the feature `(orientation +)`, `(orientation -)` and `(orientation none)`.

The feature orientation is set to the value of the argumentative evaluation + or - which has scope over the set being lexicalized. If there is no argumentative evaluation, or if the NP planning has not set realize-quantity to evaluation, it is set to none.

6.6.5. The Intersection Condition: Argumentative Constraints on NP Planning

I now consider a semantic property of quantifiers called *intersection condition* in [Barwise and Cooper 81, p.190]. This property highlights again the semantic complexity of judgment determiners, but the main point in this subsection is to:

- Justify the distinction between intension and reference in the set description presented in Sect.6.5.1.
- Identify cases where the argumentative intent impacts on NP planning.

The linguistic test corresponding to the formal definition given for the intersection condition is the following: let P_1 and P_2 be two properties, then if a determiner D satisfies the intersection condition, the sentences *There are D* P_1 P_2 *N* and *D* P_1 *N are* P_2 are semantically equivalent. For example:

There are exactly 3 interesting AI topics.
Exactly 3 interesting topics are in AI.
Exactly 3 AI topics are interesting.

These three forms are equivalent, indicating that the determiners of the form *exactly n* satisfy the intersection condition. In contrast, consider:

(1) There are many interesting topics which are in AI.
(2) There are many AI topics which are interesting.

These NPs are not equivalent, as shown, for example, by considering the following situation: a person has interest in 100 topics, AI covers 10 topics, the intersection between the interesting topics and the AI topics contains 7 elements. Then sentence (1) is probably not valid (7 topics out of 100 is not many) while sentence (2) is valid (7 out of 10 is many).[56] Note that the ''classical'' quantifiers, corresponding to the mathematical $\exists$ and $\forall$, both satisfy the intersection condition:

Every interesting topic which is in AI.
Every AI topic which is interesting.

There are some interesting topics which are in AI.
There are some AI topics which are interesting.

But the vague quantifiers, like *most*, *many* and *few*, do not satisfy it. Note that these vague quantifiers can all realize an argumentative evaluation of the number of elements in a set.

Consider now the fact that in both (1) and (2) the NP with the *many* determiner denotes the same set of individuals (the 7 topics of the intersection). The truth value of the sentences, however, is different when the scope of the determiner *many* changes from one modifier to the other. So clearly the semantic description of the set that is realized by *many interesting topics* must contain an intensional element (a point made in [Keenan and Stavi 86, 1.2]). This is a justification for the use of the features `intension` and `reference` in the set specification format shown in Sect.6.5.1.

I now explain why, since judgment determiners do not satisfy the intersection condition, the `intension` and `reference` features cannot be interchanged, and why the distinction between the two has an argumentative meaning.

---

[56]The semantics of vague determiners like *many* is discussed below. The simple model I use allows checking the validity of judgments like ''7 out of 10 is many''.

Perspective 1:

```
((cat set)
 (kind ((cat topic)))
 (cardinality 7)
 (intension ((cat user-relation) (name interest)
             (argument {^ 1}) (2 ((cat student)))))
 (reference ((cat set)
             (kind ((cat topic)))
             (cardinality 10)
             (intension ((cat class-relation) (name area)
                         (argument {^ 1})
                         (2 ((cat field) (name AI))))))))
```

Perspective 2:

```
((cat set)
 (kind ((cat topic)))
 (cardinality 7)
 (intension ((cat class-relation) (name area)
             (argument {^ 1}) (2 ((cat field) (name AI)))))
 (reference ((cat set)
             (kind ((cat topic)))
             (cardinality 100)
             (intension ((cat user-relation) (name interest)
                         (argument {^ 1}) (2 ((cat student)))))))))
```

Figure 6-24: Two perspectives on the same denotation

Consider the set defined as:

(E) $S1=\{x \in \text{TOPICS} \mid Interest(x,student) \wedge Area(x,AI)\}$

There are two possible ways to encode this definition into the set format presented in Sect.6.5.1, which are shown in Fig.6-24. In the first perspective, the `interest` property is the intension and the `area` property is the reference; the roles are reversed in the second perspective. These two perspectives can, under regular circumstances, lead to the following realizations:

Most AI topics are interesting.

Few of the topics that interest you are in AI.

In this example, the same observation of a set of topics satisfying two properties can lead to two contradictory argumentative evaluations. This indicates that the structuring of properties between `reference` and `intension` must be done by the evaluation functions producing the conceptual input to the lexical chooser: given an argumentative goal to encourage the student to take AI, perspective 1 will be selected over perspective 2 by the evaluation function. The conclusion is, therefore, that the set descriptions that must be sent to the lexical chooser must use the structured format I indicate and not the extensional, neutral format shown in (E).

Note that the set description I propose is compatible with the set descriptions proposed in [Dale 88, Sect.3.3.6], except that I extend Dale's formalism by using reference and intension, and I do not deal with quantity and mass substances, but only with countable elements.

6.6.6. Summary: Generation of Determiner Features

I have presented in this section the method used to select the syntactic features controlling the generation of the determiner sequence in an NP. I have specifically focused on the features necessary to express an argumentative evaluation of the cardinality of a countable set, *i.e.*, the features allowing the generation of judgment determiners

such as *many* and *few*. Of the 24 features used by SURGE to generate a complete determiner sequence, 5 features are directly related to the expression of argumentative evaluation. The two main features are:

- **Orientation**: is set to the value of the argumentative orientation (+ or -). If there is no argumentative evaluation, it is set to none. The class of determiners expressing an orientation of +, - and none correspond to the monotonic increasing, monotonic decreasing and non-monotonic determiners resp. identified by the linguistic test discussed on p.193.
- **Degree**: The value of degree depends on a comparison between the cardinality and a typical size, determined either by the conceptual relation of which the set is an argument, if it is a simple set (with no intension modifier). If the typical size feature is not available (for complex sets), the degree feature must be set by the content determination module (the evaluation functions).

In addition, the following features are used to generate the pragmatically strongest determiner possible when degree is set to +:

- **Total**: when argumentative orientation is positive and the cardinality of the set is known and equal to the cardinality of the reference set, then total is set to +. If argumentative orientation negative and the cardinality is 0, then total - is set. Otherwise, total is set to none.
- **Superlative**: Set to yes when the reference set is given and its cardinality known, and the cardinality is larger than half of the reference set cardinality.
- **Comparative**: Set to yes when the argumentative evaluation is of type relative. No otherwise.

When degree is set to +, the algorithm is to try to generate a total determiner if possible, otherwise a superlative determiner, otherwise a regular judgment determiner such as *many* or *a lot* is generated.

In addition, a study of a semantic property of determiners called the intersection condition has illustrated the need to distinguish between intension and reference in the conceptual set description. For a set defined by the conjunction of two properties P1 and P2, using P1 as intension and P2 as reference or vice-versa can generate argumentative determiners of opposite orientation. Therefore, the specification of an argumentative orientation on a set can serve as a constraint on which perspective to adopt on the set and therefore on NP planning. The decision to map conceptual properties to either intension or reference is performed by the content determination module (in ADVISOR II, the evaluation functions).

The main contribution of this section is the definition of a linguistically motivated method for generating judgment determiners. No other generation system to my knowledge has addressed this issue previously.

6.7. Generation of Adjectives

This section describes the lexical decisions made at the adjective level. Adjectives occur either as modifiers of NPs or as arguments of relational clauses. Adjective phrases have a simple syntax (I do not consider comparatives), and most of the time consist only of a single adjective. Different classes of adjectives, however, can be used in different contexts. In this section, I first describe the conceptual elements which can be realized as adjectives. I then review important classification dimensions for adjectives and how they impact on the use of adjectives. I derive from this analysis a list of features that are used to represent adjectives in the lexicon. Finally, I explain how these resources are mapped to adjectives.

6.7.1. Conceptual Elements Realizable by an Adjective

I first enumerate the conceptual elements which can be realized by adjectives. Traditionally, an adjective is defined as ‘‘serving as a modifier of a noun to denote a quality of the thing named, to indicate its quantity or extent, or to specify a thing as distinct from something else’’ [Webster 63]. Analysis of a corpus of advising sessions, however, shows that adjectives often loosely relate to actual properties of the objects being modified but are used more often to express a speaker’s intention or argumentative orientation. When an advisor tells a student that a *course is very hard*, he often does not refer to a property of the course, but rather expresses his evaluation of the course.

Semantic class	Adjective	Occurrences
Difficulty [24]	advanced	1
	basic	1
	challenging	1
	difficult	4
	easy	5
	hard	11
	high-level	1
Domain [8]	mathematical	2
	programming	4
	theory	1
	theoretical	1
	computing	1

Semantic class	Adjective	Occurrences
Importance [24]	important	10
	needed	1
	recommended	5
	required	5
	suggested	1
	useful	1
	valuable	1
Evaluative [10]	interesting	4
	perfect	1
	good	5
Misc [3]	traditional	1
	new	1
	interdisciplinar	1

Figure 6-25: Adjectives modifying courses in corpus

In order to assess the importance of the argumentative usage of adjectives, I performed an analysis of a corpus of 40,000 words containing transcripts of recordings of advising sessions with human academic advisors. In this corpus, I identified approximately 700 occurrences of 150 distinct adjectives. I focused the analysis on all occurrences of adjectives modifying a course, in both predicative and attributive positions. I found 69 such occurrences, of 26 distinct adjectives. Figure 6-25 shows a break down of these occurrences in semantic classes.

Of the 69 occurrences listed in Fig.6-25, 58 express a property of a course that one cannot reasonably expect to find in the knowledge-base describing courses. For example, it is problematic to describe a course as *good* or *hard* in absolute terms. For most of the occurrences, therefore, the technique of mapping from a semantic property in the knowledge-base to an adjective, as used in previous generation systems to produce attributive noun-phrases (*e.g.*, [Reiter 90; Dale 88]), would not be applicable. Most of the usages of adjectives in the corpus, therefore, correspond to an argumentative usage. For example, the advisor qualifies a course as *hard* when he wants to discourage a student from taking it. The selection of *hard* in this context is related to the underlying goal of the advisor in addition to the objective properties of the course and to the level of the student as evaluated by the advisor (the same course is not hard for all students). In the ADVISOR II domain, therefore, a large majority of adjectives are used to realize an argumentative evaluation.

There are still other usages for adjectives, corresponding to the NP modifiers already mentioned in Sect.6.5. These are:

- Describer derived from a conceptual relation used as a modifier, *e.g., a required class*.
- Classifier adjective derived from a relation deleted because it is recoverable, *e.g., a theoretical class*.
- Complement of a relational clause expressing a conceptual relation, *e.g., the class is theoretical*.

6.7.2. Linguistic Features of Adjectives

To characterize how adjectives can be used in the grammar, and which information must be stored in the lexicon for each adjective, I have identified the following issues:

- Whether an adjective can be used in attributive and predicative function, to fulfil an identifying or a classifying function. These distinctions determine where an adjective can appear in a clause or in an NP.

- Whether an adjective is scalar or non-scalar, and marked or unmarked. These distinctions determine which adjectives can be used to realize an argumentative evaluation.
- Whether an adjective is absolute or relative.
- Whether an adjective is linear.
- Whether an adjective can be intensified and if yes, by which adverbial expression.
- What range of entities the adjective can modify.

I now provide some explanation on each one of these points. In general, adjectives can occur in either attributive (X is A) or predicative position (A X) [Quirk *et al* 72, p.231]. Certain adjectives however can only be used in predicative position (*e.g.*, *mere*, *only*), only in attributive position or can have a different meaning if used in predicative or attributive position.[57] Such properties need to be encoded in the lexicon.

In [Bolinger 72, p.21], Bolinger distinguishes between *degree* and *non-degree* adjectives. Another terminology for the same distinction is *scalar* vs. *non-scalar*. In the ADVISOR II domain, *required* is an example of non-degree adjective (there is an official legal definition of what a required course is for the major), whereas *important*, *hard* or *interesting* are all degree adjectives. Non-degree adjectives cannot be used with intensifiers like *very* and cannot be used in comparative forms. This lexical classification limits the range of adjectives capable of being used for argumentative purposes.

Using different terms, linguists have distinguished between *marked* and *neutral* adjectives ([Givon 70], [Rusiecki 85, p.13 *ff*] and [Huebler 83, p.38]). To understand this distinction, consider the difference between the adjectives *hard* and *difficult*. In our corpus, *hard* was consistently used in contexts where the advisor was discouraging the student from taking a course, as in the following examples extracted from the corpus:

*Data Structure is probably the **hardest** course and you would want*
to make sure that you could handle it.
There is no law against taking Data Structures without having ...
*[pause] but it is a very **hard** course.*

In contrast, *difficult* was used in more neutral contexts, where the advisor did not commit to a particular evaluation of the course:

*I really can't tell you how **difficult** or easy they are.*
I think they're both at the same level and I don't think there's
*much difference in terms of what's easier and more **difficult**.*

Hard and *difficult* convey a very similar information on the course. However, *hard* is marked, while *difficult* is neutral. In general, a test to identify neutral adjectives is to form a question *how A is it?*. For example, *how expensive is it?* does not prejudge of the price, while *how cheap is it?* expresses a judgment that the object is cheap. Note however that while *difficult* tends to be used as a neutral adjective, it is nonetheless scalar, and when it is intensified, it becomes marked (like in *it is a very difficult class*). This lexical property distinguishes among adjectives conveying the same information those that can be used to convey an argumentative meaning.

Many linguists have distinguished between *absolute* and *relative* adjectives [Bartsch 89; Huebler 83, p.37]. The meaning of relative adjectives depends on the object being modified (a *small* elephant is a *big* animal) whereas absolute adjectives keep the same denotation for all objects they modify (a *red* box is as *red* as a *red* book). For relative adjectives, an evaluation norm needs to be identified. This norm can be explicitly stated as in *Data Structures is the hardest **of the undergrad courses*** or *this course would be perfect **for you.*** But it can also be left implicit as in *this course is fairly advanced* where the evaluation norm determining what is *advanced* depends on the model the speaker has of the student. In the ADVISOR domain, I have found that relative adjectives depend not only on the object being modified (a *good* course is not good in the same sense as a *good* meal) but also depend on a model of the hearer: a *challenging* course for an undergrad could be easy for a graduate student, a programming project could be very difficult for a student lacking programming experience.

[57] For example *old* in *an old friend* is the opposite of *new*, whereas in *My friend is old* it is the opposite of *young*.

Certain adjectives can be presented as *absolute* in surface. For example, *interesting* was consistently used in the corpus without any post-modifying complement:

What is that course? It looked very interesting

It would be an interesting course. I mean, I think
Mathematical Logic is pretty [pause] interesting.

In contrast, *good* was always used with a complement explicitly relativizing its meaning:

So that might be a good class for you to take next semester if you take AI this semester.

If you're good at math - that might be a good course to take.

Note that this distinction is only at the surface: there is good reason to consider *interesting* as a relative adjective in the semantic sense introduced above and many semantically relative adjectives do not require or prohibit an explicit complement at the surface. This property of *good* and *interesting* is therefore unpredictable from their semantics. But it constrains the way these adjectives can be used. An interesting generalization of any construct of the form *X is A* to an underlying form *X is A for an x* for all relative adjectives is presented in [Ludlow 89], but I prefer to leave the behavior of each relative adjective specified in the lexicon.

In [Klein 80, p.6], Klein introduces a distinction between *linear* and *non-linear* adjectives. The definition he provides is:

> Whenever *c* is a context of use, and NP_1 and NP_2 denote individuals within the sortal range of A, then the sentence *NP_1 is A-er than NP_2* has a definite truth value in *c*.

This definition only applies to relative adjectives. Intuitively, a relative adjective is linear if there is a single criterion determining how to evaluate an object on the scale the adjective denotes. For example, *tall* is a linear adjective, ''which linearly orders any set X of vertically extended objects: if $u,v \in X$, then either *u* is taller than *v*, or *v* is taller than *u*, or *u* is exactly as tall as *v*.'' Most adjectives in English however are non-linear. Consider for example the adjective *difficult* in the ADVISOR II domain. This adjective is ''associated with a number of criteria, and these fail to constitute a necessary and sufficient set of conditions'' for difficulty [Klein 80, p.7]. To simplify, assume that a class can be difficult either because it has a heavy workload, or because it covers conceptually complex topics. Suppose that AI has a heavy workload, and Analysis of Algorithms covers complex topics. Then the truth-value of *AI is more difficult than Analysis of Algorithms* has no definite truth value. This distinction is important to determine the semantics of adjectives and to which conceptual elements they are related.

At the semantic level, the lexicon specifies the mapping from conceptual scales to the adjectives that can express them. The non-linearity of an adjective like *difficult* is not represented in the lexicon (*difficult* is mapped to the single scale `difficulty`), but is explained by the set of topoi where `difficulty` appear in right-hand position. In addition to mapping adjectives to scales, the lexicon also must specify which objects can be modified by which scales. For example, the meaning of *hard* is different when modifying a course, a material or a liquor. Different scales correspond to each one of these classes.

Similar to these selection restrictions but at the lexical level, lexical affinities or collocations [Smadja 91a] can constrain what words can be used along with adjectives. For example, a course can be *strongly recommended* or *very important* [Bolinger 72, pp.21-57]. The choice of the intensifier is constrained by the adjective. Such lexical affinities are captured in the lexicon.

Figure 6-26 shows an example of lexical entry for the adjective *hard*. For the semantic section of this entry, the `object` feature contains the semantic class of the object being modified. The `alt` construct lists the semantic classes compatible with the adjective. For each type of object, the argumentative scale triggered by *hard* is different. For *hard*, the Webster dictionary lists 13 different meanings corresponding roughly to different scales. This semantic description needs to be adapted to different domains.

```
((cat adjective)
 (lex "hard")
 ;; Compatible semantic classes that can be modified
 (object ((alt (((cat course))
                ((cat material))
                ((cat liquor))
                ...))))

 ;; Depending on semantic class of object, semantic scale triggered by the adj
 (alt (
   ((object ((cat course)))
    (ao ((scale difficulty) (orientation +))))
   ((object ((cat material)))
    (ao ((scale pressure-resilience) (orientation +))))
   ((object ((cat liquor)))
    (ao ((scale alcoholic-concentration) (orientation +))))
   ...))

  ;; No collocation constraints on intensifier: use default
  (intensifier nil)

  (degree yes)
  (marked yes)
  (relative yes)
  (require-complement no)

  ;; can be used both in predicative and attributive position
  (predicative yes)
  (attributive yes))
```

Figure 6-26: A lexical entry for the adjective *hard*

6.7.3. Mapping Conceptual Elements to an Adjective

Two types of elements can be mapped to an adjective: argumentative evaluations and conceptual relations. The mapping process from conceptual relations to adjectives has been presented in Sect.6.5 (for both predicative and non-predicative adjectives). I now discuss how argumentative evaluations can be mapped to adjectives.

An argumentative evaluation is characterized by a scale and an orientation. It can be realized by being merged with other meaning-conveying linguistic elements or by a linguistic element entirely devoted to its expression. If the expression is only realizing the argumentative evaluation explicitly, the lexicon entry for the scale must provide enough information to produce such a linguistic element in its entirety.

An argumentative evaluation is a sort of predication, which can be realized by a clause. In the lexicon, the scale category describes the mapping between scales and these clauses. In most cases, the clause is an attributive clause of the form *X is P*, *e.g.*, *AI is difficult*. In some cases, it can also be a clause of a different type: *AI has a lot of programming* for the `programming` scale, or *AI requires a lot of work* for the `workload` scale. When the clause is attributive, the adjective expresses the scalarity of the evaluation. When another type of clause is used, the scalar element is another constituent of the clause - an NP marked with a judgment determiner in *AI has a lot of programming*, or a verb marked with an adverb in *AI advances quickly*. I call the linguistic element which expresses the scalarity of the clause the *argumentative focus* of the clause. For each scale, the lexicon provides a clause expressing an evaluation of an entity on the scale. Two examples are shown in Fig.6-27.

The top of the figure shows the structure of the input representing scales. It is an FD of category scale, with features name, identifying the scale, evaluated, identifying which entity is evaluated by the scale, and orientation. The bottom of the figure shows how scales are mapped to clauses. The first entry describes the clause realizing the scale of difficulty: *X is difficult / hard / easy*, where *X* is the evaluated entity. The entry distinguishes between the

```
;; An input argumentative evaluation: AI is difficult.
(setf arg-ev1
  '((cat scale) (name "programming")
    (evaluated ((cat class) (name "AI")))
    (orientation +)))

;; Lexicon entries for scales
(def-alt scales (:index name)
  (
   ;; X is difficult/hard/easy
   ((name "difficulty")
    (+
     ((pred ((process ((type ascriptive)))
             (scalar {^ participants attribute})
             (participants
              ((carrier {^4 evaluated})
               (attribute ((cat adj)
                           (lex ((ralt ("difficult" "hard")))))))))))
    (-
     ((pred ((process ((type ascriptive)))
             (scalar {^ participants attribute})
             (participants
              ((carrier {^4 evaluated})
               (attribute ((cat adj) (lex "easy")))))))))

   ;; X has /a lot of/little/ programming
   ((name "programming")
    (pred ((cat clause)
           (process ((type possessive)))
           (scalar {^ participants possessed})
           (participants
            ((possessor {^3 evaluated})
             (possessed ((cat common) (lex "programming")
                         (countable no)
                         (orientation {^4 orientation}))))))))

    ...))
```

Figure 6-27: Lexicon entry for the scales `workload` and `difficulty`

(orientation +) and (orientation -) cases, and provides a different mapping for each. The mapping specify that the scale is realized by an ascriptive clause, whose carrier role is filled by the evaluated element and attribute role is filled by the scalar adjective *difficult*, *hard* or *easy*. In addition, the feature `scalar` points to the attribute role, indicating that this attribute expresses the scalarity of the evaluation. In the second entry, for the programming scale, I do not distinguish between a + and a - orientation, instead, the orientation is conflated into the linguistic constituent expressing the scalarity. The clause is a possessive relation of the form *X has /a lot of / little / programming*. The scalar feature now points to the possessed role, which is an NP.

Only the first one of these two scales can be mapped to an adjective, because it is realized by an ascriptive clause which can be reduced to a describer. The second scale cannot be reduced to an adjective.

When an adjective is generated to realize an argumentative evaluation, I keep the option of inserting an intensifier to emphasize its scalar nature. The default intensifiers are *very* and *quite*. The same politeness criteria used to distinguish between *should* and *could* at the clause level are used to choose between *very* and *quite*. For evaluations that are face-threatening, *quite* can be used to create distance between the utterance and the hearer; for evaluations that are non face-threatening, *very* can be used. Of course, both intensifier (*very*) and detensifier (*quite*) forms are replaced by the proper collocate of the head adjective when necessary. The decision to use an adjective modifier or not is made randomly.

6.7.4. Conclusion: Constraints on Lexicalization of Adjectives

I have presented in this section the lexical representation of adjectives and its usage. I have distinguished between the following classes of adjectives:

- attributive and predicative adjectives
- scalar and non-scalar
- marked and unmarked
- absolute and relative
- linear and non-linear

Membership in these classes determines the syntactic behavior of an adjective and which adjectives can serve to realize an argumentative evaluation. Most of the adjectives used in the ADVISOR II domain have been found to be used argumentatively, so I have focused on how argumentative evaluations are mapped onto adjectives:

- Each scale is mapped to a clause expressing the evaluation.
- The clause is transformed into an appropriate modifier using the techniques described in Sect.6.5 on NPs.

Finally, adjectives used in argumentative function can be modified by an intensifier or a detensifier. The choice between intensification and detensification is based on politeness principles. The choice to use an adjective modifier is made randomly.

The main contribution of the lexicalization procedure at this level is to give proper consideration to the scalar nature of adjectives and make adjectives realize argumentative evaluations as opposed to only knowledge-based properties of objects.

6.8. Generation of Connectives

This section describes how connectives are selected in a clause-complex. I focus on the connectives *although*, *but*, *because*, *since* and *so*. These five connectives all express an argumentative connection between clauses, with *because*, *since* and *so* expressing support, and *but* and *although* expressing contrast. Since connective choice is mainly related to paragraph organization, I follow a slightly different format in this section than in the four previous ones: I first briefly review previous work on the description of connectives. I then identify distinctions between the five connectives I consider, and derive from these distinctions a set of features which must be present in the input to control the selection of a connective. Lexical entries for some connectives are finally presented, illustrating how an input paragraph plan can be lexicalized to include connectives.

6.8.1. Previous Work on Connectives

The most basic constraint on connection is often referred to as homogeneousness condition: two propositions can be conjoined if ‘‘they have something in common.’’ Which features of the conjuncts must be homogeneous is a difficult question: [Chomsky 57, p.36] stated a constraint on syntactic homogeneousness (conjuncts must be ‘‘of the same type’’); a purely syntactic constraint is, however, largely insufficient to satisfy the needs of a text generation system, since the decision to conjoin must be made before the syntactic structure of the conjuncts is determined. [Lakoff 71] proposed a semantic approach to the problem of homogeneousness: conjuncts must have a ‘‘common topic’’ for conjunction to be possible (p. 118). Based on this definition of homogeneousness, she distinguished between a ‘‘semantic’’ meaning of ‘‘but’’ (to express a semantic opposition) and a pragmatic usage of ‘‘but’’ (to deny expectations), for cases which would not satisfy the homogeneousness constraint (*e.g.*, ‘‘John is rich but dumb’’). Such a distinction between a semantic and a pragmatic analysis of connectors is criticized in [Abraham 79, p.104] [Lang 84, pp172ff] and [Ducrot *et al* 80]. Lang (1984) presents a general semantics for conjunction that does not distinguish between pragmatic (or contextual) and semantic levels. Lang attributes to conjunctions an operative semantics: conjunctions’ meanings are sets of ‘‘instructions’’ for ‘‘carrying out certain mental operations’’ (p.

96)[58]. The meaning of connectors is a ‘‘program’’ that controls how a ‘‘common integrator’’ can be constructed from the meaning of each conjunct. In this work, I use a similar approach for the definition of connectives, but, since I work on generation (as opposed to interpretation), I describe the meaning of connectives as sets of constraints that must be satisfied between the conjuncts as opposed to ‘‘instructions.’’

Work on the structure of discourse [Cohen 84; Reichman 85; Grosz & Sidner 86] has identified the role of connectives in marking structural shifts. This work generally relies on the notion that hearers maintain a discourse model (which is often represented using stacks). Connectives give instructions to the hearer on how to update the discourse model. For example, ‘‘now’’ [Hirschberg & Litman 87] can indicate that the hearer needs to push or pop the current stack of the model. When used in this manner, connectives are called ‘‘cue (or clue) words.’’ This work indicates that the role of connectives is not only to indicate a logical or conceptual relation, but also to indicate the structural organization of discourse. The distinction between cue and non-cue usages is an important one, and I also attempt to capture cue usages, but the structural indication (which often has the form of just push or pop) under-constrains the choice of a cue word - it does not control how to choose among the many markers indicating a *pop*.

Halliday [Halliday 85] proposes that the connection between clauses can be described on three dimensions: *taxis*, *expansion* and *projection*. This model is implemented in the *Nigel* system [Mann & Matthiessen 83b]. It provides a fine-grained classification of a broad set of connectives. However, labels used to describe the type of relation between two propositions within the *expansion* system are similar to rhetorical relations and precise definitions of these relations, to date, have tended to be subjective.

Like Halliday, I also attempt to provide a fine-grained characterization of connectives and the model presented here has features that are similar to Halliday’s *taxis* and *projection* systems. However, the use of argumentative features avoids a reliance on unanalyzed rhetorical relations.

This work is influenced by work in pragmatics on implicature [Levinson 83; Karttunen & Peters 79] which proposed a two-level representation of utterances (propositional content and implicatures). It is also based on a ‘‘multi-dimensional’’ description of utterances and describes connectives as devices acting on each pragmatic dimension.

6.8.2. Distinction Support vs. Contrast

The first distinction in the group of five connectives I consider is between those expressing support (*because*, *since* and *so*) and those expressing contrast (*but* and *although*). To account for this difference, I use an argumentative analysis. A support connective *c* is used in *PcQ* if *P* is presented as supporting a conclusion *R* and *Q* realizes the evaluation *Q*. *P* is presented as supporting *Q* if it realizes an evaluation *L* and there exists a topos of the form *<+ L, + R>*. For example, in *AI has many assignments so it could be difficult*, *P* (*AI has many assignments*) realizes the evaluation *L* `<cardinality AI assignments, +>` and *Q* (*AI could be difficult*) realizes the evaluation *R* `<difficulty AI, +>`. The topos `<+ cardinality X assignments, + difficult X>` can be instantiated into */L, R/* so the support connective *so* can be used. The other support connectives *because* and *since* work in the other direction, where *P* realizes *R* and *Q* realizes *L*.

To use a contrast connectives in *PcQ*, the two clauses must express argumentative evaluations *L* and *R* such that a topos of the form *<+L, -R>* or *<-L, +R>* exists. For example, in *AI has many assignments but it is quite easy*, the activated topos is `<+ cardinality X assignments, + difficulty X>` and *Q* realizes the evaluation `<- difficulty X>` producing the contrastive effect. In the case of *but*, the contrastive effect can be *indirect*. For example, in *AI can be difficult but it is very interesting*, *two* topoi are involved: *P* realizes the evaluation *<+ difficulty X>* and activates the topos */+ difficulty, - take/*, and *Q* realizes the evaluation *<+ interest X>* and activates the topos */+ interest, + take/*. Here the contrast is between the two implicit conclusions */+ take/* and */- take/*. In terms of evaluations, I say that *but* can connect two composite evaluations, while the other ‘‘direct’’ connectives connect a composite evaluation to a simple evaluation.

[58] A similar operative approach is advocated in [Ducrot 83]

6.8.3. Distinction *but* vs. *although*: Functional Status

But and *although* can be distinguished by their influence on the discourse structure in which they are embedded. I draw upon a theory of conversation organization common in conversation analysis [Sinclair & Coulthard 75; Taylor and Cameron 87; Roulet *et al* 85; Moeschler 86] to explain this distinction. The model describes conversation as a hierarchical structure and defines three levels of constituents: *speech acts*, *move* and *exchange*. A *move* corresponds to a turn of a speaker in a conversational *exchange* between two or more speakers. It is made up of several *speech acts*. In the structure of a move, one speech act is *directive*; all others are *subordinate* - they modify or elaborate the *directive* act [Roulet *et al* 85]. Intuitively, the *directive* act is the reason why the speaker started speaking. It constrains what can follow the move in the discourse. While a move may consist of several subordinate speech acts in addition to the directive act, the directive controls the possibilities for successive utterances. Thus, it determines what is accessible in the structure of the preceding discourse.

To see how this characterization of discourse can explain the distinction between *but* and *although*, consider the following examples:

(1) * He failed the exam,
although he is smart. Let's hire him.
(2) He failed the exam,
but he is smart. Let's hire him.

In both (1) and (2), the first sentence expresses a contrastive relation between two propositions. But, the full sequence (2) is coherent, whereas the sequence (1) sounds peculiar in most situations. This can be explained by the fact that in ''P but Q'' Q has directive status while in ''P although Q,'' Q has subordinate status. In (2) then, *he is smart* has directive status, whereas in (1) it is subordinate. Therefore, the argumentative orientation of the complex sentence as a whole in (1) is the argumentative orientation of *he failed the exam* and it is the argumentative orientation of *he is smart* in (2). The conclusion (*let's hire him*) is only compatible with *he is smart.*

This distinction is similar to Halliday's taxis system (the classic subordinate/coordinate distinction) but operates at a different level. Although *but* is a conjunction, meaning that P and Q have the same syntactic status, P and Q have a different influence on the following discourse. I, therefore, require the input to the surface generator to indicate the ''point'' of a move, but to leave the syntactic status of each proposition unspecified. This more delicate decision is made by the surface generator.

This same distinction explains the difference between *since* and *so*. A pattern *Q so P* has Q as directive act, whereas *Since Q, P* has P as directive act.

6.8.4. Distinction *because* vs. *since*: Polyphonic Features

Because and *since*[59] have the same argumentative behavior and give the same functional status to the propositions they connect. Their different usages can be explained using Ducrot's theory of *polyphony* [Ducrot 83]. Ducrot distinguishes between the *speaker* and the *utterers*: in an utterance, some segments present beliefs held by the speaker, and others present beliefs reported by the speaker, but attributed to others - the utterers.

Using this theory, the difference between *because* and *since* is as follows: in the complex *P since Q*, the segments *P* and *Q* can be attributed to different utterers (*since is polyphonic)*, whereas in *P because Q*, they must be attributed to the same utterer (*because* is monophonic).

Others have described *because* and *since* by noting distributional differences such as:

1. To answer a *why* question, only *because* works:

   A: Why did Peter leave?
   B: <u>Because</u> he had to catch a train.

[59] I consider only the causal meaning of *since* here

*B: *Since he had to catch a train.*

2. *Because* has a tendency to follow the main clause while *since* has a tendency to precede it [Quirk *et al* 72, 11.37].

3. *because*-clauses can be the focus of cleft sentences [Quirk *et al* 72]:

It is because he helped you that I'm prepared to help him.
**It is since he helped you that I'm prepared to help him.*

The given/new distinction gives one interpretation of these differences: *because* introduces new information, whereas *since* introduces given information (where given is defined as information that the listener already knows or has accessible to him [Halliday 85]). Halliday also indicates that, in the unmarked case, new information is placed towards the end of the clause. And indeed *because* appears towards the end, the unmarked position of new information, and *since* towards the beginning. *Because* can be the focus of an It-cleft sentence which is also characteristic of new information (cf [Prince 78] for example). *Because* can answer a why-question, thus providing new information to the asker. Presenting given information in response could not serve as a direct answer.

There are many different types of given information, however [Prince 81]. Polyphony is one type of given information but it adds an additional parameter: each piece of given information is attributed to a particular utterer. That utterer can be one of the speakers (this is similar to indirect speech), or it can be a mutually known previous discourse. The ability to distinguish how the *since* clause is given (i.e., which utterer contributed it) is crucial to correct use of sentences like (3).

From a father to his child:
(3) Since you are so tired, you must sleep.

In (3), the speaker presents the hearer as the source of *you are tired*, and uses the fact that the hearer has previously uttered this sentence as the argument for *you must sleep*. If the hearer is not the source of the sentence, this strategy cannot convince him to go to sleep. Given/new in this case is, therefore, a polyphonic distinction, and polyphony provides an added dimension to the distinction.

In summary, *because* and *since* have the same argumentative and functional status definitions, but they have different polyphonic definitions. *Because* requires *P* and *Q* to have the same utterers, while *since* does not.

6.8.5. Input Necessary to Select a Connective

In the previous subsections, I have identified the following factors to distinguish between connectives:

- The argumentative relation expressed by the connectives. Each connective indicates that the conjuncts are related by a certain pattern of argumentative relations, and thus certain topoi are activated by the usage of the connective.
- The functional status of each conjunct in a discourse segment. I distinguish between directive and subordinate segments, the directive segment expresses the main point of the whole segment, and its argumentative orientation of the whole.
- Polyphonic features. Each discourse segment can be attributed to different utterers, which can each be endorsed by the speaker or not. An utterer is defined as a coherent source of argumentative evaluation. So, for example, whenever a contrastive utterance is produced, I assign each part of the contrast to different utterers, with the utterer of the directive act endorsed by the speaker.

I therefore use the set of features shown in Fig.6-28 to represent an input to FUF which encodes enough information to allow a non-arbitrary selection of connectives (some constituents of this FD are elided for clarity). The toplevel structure of the FD encodes the discourse segment structure. Two categories are used in this representation: discourse-segment and utterance. A discourse-segment is a complex structure containing a directive elements and a subordinate segment. An utterance is the leaf of the discourse structure, and is realized by a clause. The discourse-segment FD, therefore, describes a binary tree which can encode arbitrary tree-structures. The tree corresponding to Fig.6-28 is shown in Fig.6-29.

```
(def-test connective1
  "AI has many assignments which require writing papers.
   You have little experience in writing papers.
   So it could be difficult."
  ((cat discourse-segment)
   (subordinate
    ((cat discourse-segment)
     (directive ((cat utterance)
                 (ao ((cat topos)
                      (justification <AI has 6 assignments>)
                      (left ((cat evaluation) (orientation +)
                             (scale ((name "cardinality")))))
                      (right ((cat evaluation) (orientation +)
                              (scale ((name "difficulty")))))))
                 (pc ((semr ((cat relation) ...))))
                 (utterer ((name u1) (speaker yes)))))
     (subordinate
      ((cat utterance)
       (pc ((semr ((cat relation)
                   <you have no experience writing papers>)))))))
   (directive
    ((cat utterance)
     (ao ((cat evaluation) (orientation +)
          (scale ((name "difficulty")))
          (evaluated ((index ai1)))))
     (utterer ((name u1) (speaker yes)))))))
```

Figure 6-28: Features at the connective level

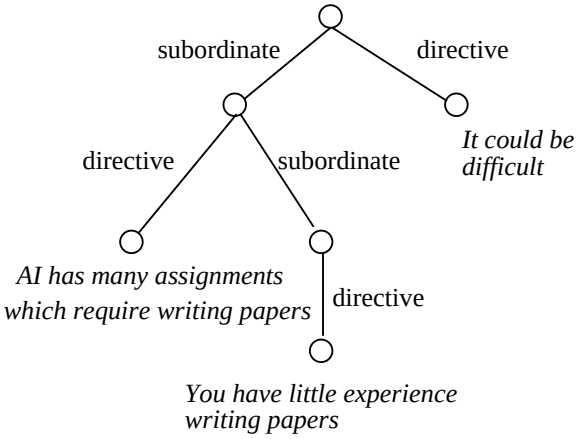


Figure 6-29: Tree structure analysis of a discourse segment

An FD of category utterance contains three main features: AO (argumentative orientation), PC (propositional content) and Utterer. Each one corresponds to a different perspective on the function of an utterance. AO describes the argumentative function of the utterance. It can be either a topos (with features left, right and justification, as discussed above) or simply an evaluation (with features orientation, scale and evaluated). The PC feature describes the informational function of the utterance. It encodes the information conveyed by the utterance. It is an FD of category clause. Finally, the Utterer feature encodes the polyphonic feature: which utterer is producing the utterance, and whether the speaker endorses the argumentative position of the utterer or not.

6.8.6. Selecting Connectives in Discourse Segments

I describe now how connectives are selected in a discourse segment represented in the format shown in Fig.6-28. Connectives are represented in the lexicon as relations between utterances, that is, a partially filled discourse-segments. Figure 6-30 shows the lexical entry for the connectives *so* and *but*. The constraints enforced by the entry for *so* at the top of the figure are that, in a conjunction *P so Q*:

- *P* realizes a composite evaluation */L, R/*.
- *Q* realizes a simple evaluation *R* (this is enforced by the equation `({^ subordinate ao right} {^3 directive ao})`).
- A topos */+L, +R/* can be found in the grammar of topoi (this is indicated by putting the feature (cat topos) in the FD, forcing the subconstituent to be unified with the grammar of topoi).
- The utterers of *P* and *Q* are the same, and they are both endorsed by the speaker.
- The order of the elements is fixed to *PcQ* (as indicated by the pattern constraint).

The bottom of Fig.6-30 shows the lexical entry for the connective *but*. This entry again encodes a set of constraints that must hold between two discourse segments *P* and *Q* so that the connection *P but Q* can be generated. The constraints encoded for *but* are the following:

- Argumentative contrast: the contrast can be either direct or indirect. A direct contrast corresponds to the case where *P* realizes a composite evaluation */L1, +R/* and *Q* realizes the simple evaluation */-R/*. An indirect contrast corresponds to the case where *Q* also realizes a composite evaluation */L2, -R/*. In both cases the constraint is that the signs of the evaluations on *R* be opposite, +/- or -/+.
- A polyphonic constraint: P and Q are attributed to two different utterers, and only Q is endorsed by the speaker.
- Order and functional status: the pattern *P but Q* is not movable (* *but Q, P*) and *P* has subordinate status and *Q* is directive.

The discourse structure encoded by the directive / subordinate constituents is built by the paragraph planner by combining the output of several evaluation functions. I have not looked at the issues involved in paragraph planning. The ADVISOR II system uses an ad-hoc technique discussed in Chap.7 to do paragraph planning. Given this discourse structure in input, the lexicon can determine which connectives to use using the connective descriptions shown in this section.

Open problems include how to determine whether a connective is required. For example, in [Danlos 87a], Danlos studied the expression of the causality relation, and found that in most cases, no connective was used to express it. Similarly, [Lascarides & Oberlander 92] propose a theory based on abduction to determine when temporal connectives are required and when simple sequencing is sufficient, as in *Max entered the office. John greeted him*, as opposed to *Max fell and then John pushed him*. Such a theory determining when argumentative connectives are required in the first place would be a great enhancement to the selection method described in this section.

6.8.7. Conclusion: Constraints on Selection of Connectives

I have identified in this section three classes of constraints determining the selection of connectives:

- Discourse structure and functional status: the distinction between directive and subordinate elements in a discourse segment affects which connectives can be used.
- Argumentative relations: the five connectives I have studied express two categories of relation - support and contrast. These relations can be defined in terms of the argumentative orientation realized by each of the conjunct.
- Polyphonic features: the notion of utterer is useful to distinguish between the different sources combined into a discourse, and distinguish between connectives like *because* and *since*.

```
((cat discourse-segment)
 (connective ((lex "so") (cat conjunction)))

 ;; Argumentative support
 ({^ subordinate ao right} {^3 directive ao})
 (subordinate ((ao ((cat topos)))))
 (directive ((ao ((cat evaluation)))))

 ;; Same utterer
 ({^ subordinate utterer}  {^2 directive utterer})
 (subordinate ((utterer ((speaker yes)))))

 ;; Order PcQ: P is subordinate, Q is directive.
 (pattern (subordinate connective directive)))


((cat discouse-segment)
 (connective ((lex "but") (cat conjunction)))

 ;; Argumentative contrast
 (alt
  (;; Direct contrast P/L,+R/ Q/-R/ or P/L,-R/ Q/+R/
   (({^ subordinate ao right scale} {^4 directive ao scale})
    ({^ subordinate ao right evaluated} {^4 directive ao evaluated})
    (subordinate ((ao ((cat topos)))))
    (directive ((ao ((cat evaluation)))))
    (alt ((({^ subordinate ao right orientation} +)
           ({^ directive ao orientation} -))
          (({^ subordinate ao right orientation} -)
           ({^ directive ao orientation} +)))))

   ;; Indirect contrast P/L1,+R/ Q/L2,-R/ or P/L1,-R/ Q/L2,+R/
   (({^ subordinate ao right scale} {^4 directive ao right scale})
    ({^ subordinate ao right evaluated}
     {^4 directive ao right evaluated})
    (subordinate ((ao ((cat topos)))))
    (directive ((ao ((cat topos)))))
    (alt ((({^ subordinate ao right orientation} +)
           ({^ directive ao right orientation} -))
          (({^ subordinate ao right orientation} -)
           ({^ directive ao right orientation} +)))))))

 ;; Different utterers
 (subordinate ((utterer ((speaker no)))))
 (directive ((utterer ((speaker yes)))))

 ;; Order PcQ: P is subordinate, Q is directive.
 (pattern (subordinate connective directive)))
```

Figure 6-30: Lexical entry for the connectives *so* and *but*

6.9. Constraints on Lexical Choice

This chapter has presented a lexical choice method capable of realizing the same conceptual elements at different linguistic levels (a capability I have called cross-ranking), and capable of merging several conceptual elements into a single linguistic element. I have listed the decisions made by the lexical chooser at the following linguistic ranks:

- Clause
- Noun group
- Determiner

- Adjective
- Connective

In each case, I have identified which conceptual elements can be realized at each rank and briefly presented the syntactic features expected by SURGE, the syntactic realization component, to generate a constituent at this rank. This defines input and output for the lexical chooser. I have put the focus on the definition of a conceptual input which makes cross-ranking and merging possible, and emphasized the lexical variety which can be achieved by the lexical chooser.

Finally, having defined both input and output of the lexical chooser, I have described the mapping from input to output, listing external influences on this process when necessary. In the lexicalization process, I distinguished between syntagmatic and paradigmatic decisions. Syntagmatic decisions determine the structure of a linguistic constituent when its lexical head is selected. Syntagmatic decisions are particularly important at the clause and NP level, and I have identified the tasks of clause planning and NP planning as crucial sources of paraphrasing power. Paradigmatic decisions determine the selection of a lexical entry among a set of substitutable entries. Examples of paradigmatic decisions include the selection of *hard* vs. *difficult*, or *but* vs. *although*.

The main contributions of the method presented in this section are:

- The definition of a conceptual input to lexical choice which is not committed to any *a priori* linguistic realization decisions.
- The definition of cross-ranking realization: argumentative evaluations can be mapped to different levels of the linguistic structure, and relations can be realized as either main clauses, embedded clauses, relative clauses, noun in a noun-noun modification, adjectives or PPs.
- The definition of merging in realization: several conceptual elements can be merged onto a single linguistic element. For example, two relations can be merged into a composite process clause, and an argumentative evaluation can be merged into a verb in a clause and force the selection of a verb conveying a connotation.
- The definition of clause and NP planning, showing their impact on paraphrasing power; and the identification of pragmatic constraints on these two processes.

More specific contributions are:

- A compositional derivation of non-predicative modifiers in the noun-group, allowing the production of compounds like *AI homework* without requiring them to be all entered in advance in the lexicon.
- The production of inexact and vague determiners (also known as judgment determiners) like *many* and *few* based on the argumentative orientation of the speaker.
- The production of scalar adjectives, like *interesting* and *difficult*, also based on the argumentative intention of the speaker.
- The production or argumentative connectives, like *but* and *so*.

All of these are capabilities which were not provided by previous generators. The lexical chooser presented in this chapter, therefore, extends the range of linguistic devices covered by automatic generation and authorizes the production of more fluent text without resorting to ad-hoc procedures or canned text.

Chapter 7
Advisor II System Implementation

The ADVISOR II system is designed to provide explanations to university students planning their schedule for a semester. It is a variant of the ADVISOR system developed at Columbia in 1982-1986 [McKeown 88]. ADVISOR II was implemented to illustrate the use of FUF and SURGE in a complete generation system. The ADVISOR II domain was selected in part because the ADVISOR system was available and had provided experience into the development of an explanation component for an expert system and into the modeling of the knowledge base for the advising domain, and in part because the advising domain naturally requires argumentation and the use of evaluative language. The original ADVISOR system included a parser, plan inference module, rule-based expert system and explanation generator. ADVISOR II is a reconstruction of the explanation module only. It is not connected with the parser and plan inference module, and it includes its own version of the expert system, built around evaluation functions. ADVISOR and ADVISOR II are, therefore, complementary but they have not been merged into a single complete system.

ADVISOR II accepts three types of questions as input:

- Should I take <c>?
- How is <c>?
- How <att> is <c>?

where <c> is a class and <att> is an attribute of a class, for example *difficult*. ADVISOR II relies on a user model indicating which courses the user has taken, his interests and skills. This information is acquired by filling in a user profile before a consultation. The following slots constitute the user profile:

- credits: number of credits earned by the student in the program.
- taken: list of classes already taken by the student, with for each, optionally, the grade received.
- taking: list of classes the students has already selected for the coming semester.
- concentration: area of concentration of the student (software, hardware, AI or theory).
- degree: highest degree held by the student.
- degree-candidacy: degree aimed for by the student.
- interest: list of topics in which the student has expressed interest.
- experience: list of topics and types of assignments in which the student has experience.

All this information is gathered at once from the user by asking the user to fill in a form. In the original ADVISOR system, some of this information was dynamically inferred by the system, using plan inference techniques (for example interest).

The main emphasis of the system is on lexical choice and surface realization. The other components of the system are fleshed out to demonstrate how the input to the lexical chooser presented in Chap.6 can be produced in a complete system.

7.1. Size of the System

ADVISOR II uses four knowledge sources: knowledge base, evaluation functions, lexicon and grammar. The size of each of these components is characterized as follows:

- **Knowledge base**: written in CLASSIC, a KL-ONE type of knowledge representation. Size of source files is 500 lines (19K). The knowledge base contains 82 concepts (44 domain independent, *e.g.*, `place` or `physical-object`, and 38 domain specific, *e.g.*, `degree` or `topic`), 54 relations (23 domain independent, *e.g.*, `cause`, and 31 domain specific, *e.g.*, `assignments`), and 57 instances.
- **Evaluation functions**: written in COMMON LISP and FUF. Size of source files is 1100 lines (35K). 13 evaluation functions, 8 scales and 12 canonical topoi are defined.
- **Lexicon**: written in FUF. Size of source files is 3,300 lines (111K). The lexicon contains entries for 10 relations, 70 individuals and includes 256 words. The lexicon contains 160 choice points (alts).
- **Realization grammar**: written in FUF. Size of source files is 4,750 lines (163K). The grammar contains 370 choice points (alts).

In addition to these knowledge sources, the following processing components have been developed:

- **FUF**: written in COMMON LISP. Size of source files is 9,200 lines (336K).
- **Paragraph planner**: written in COMMON LISP. Size of source files is 500 lines (16K).

Processing time is an average of 4 minutes for the generation of a paragraph of 6 clauses running on 1991 class workstation (SUN SPARCstation 2, 28 MIPS equipped with 16Mbytes of memory). Monitoring of the code indicates that more than 70% of the processing time is spent in a single function in the FUF interpreter (`go-down-path`, the function traversing an FD to get the value of a path). Optimization of this function has not been performed but I have good hope that it can be, using a unification algorithm based on the union-find algorithm, as described in [Ait-Kaci 84]. Before this bottleneck is optimized, an easier potential optimization is to extract sub-fds from the total-fd when moving from lexical choice to syntactic realization.[60]

7.2. System Architecture

This section presents the overall architecture of the ADVISOR II system. Each component is presented in more detail in the following sections. Figure 7-1 is a flow diagram of the system. Rectangle boxes represent knowledge sources, elliptic boxes are processing components, and shaded boxes are the the data structures built by each component. Note that the current version of the system is not complete as it does not include a natural language parser and it requires manual intervention during the evaluation stage in the content planner.

The evaluation system is the component that evaluates a class along one or several scales, based on information about the class and about the user. When several scales are involved, the evaluation system also combines the evaluations into a single evaluation. The evaluation system takes as input a class and a scale. It first invokes all the evaluation functions on the class. Each evaluation function evaluates the class on a scale, based on observations in the knowledge base and the user model. These evaluations are then fed to the topoi base, which is activated as a rule base. Forward chaining through the topoi produces a list of argumentative chains all ending with the target scale. The output of the evaluation system is a list of evaluation chains with for each chain a conceptual network extracted from the knowledge base and user model justifying the evaluation. The evaluation system accesses the knowledge base, user model, topoi base and evaluation functions to perform its task. The knowledge base and user model are described first in Sect.7.3, and the operation of the evaluation system is described in Sect.7.4.

---

[60] The main penalty in efficiency is paid when traversing deeply embedded FDs and FDs with heavy cross-linking because `go-down-path` is not optimized, and it takes time linear in the length of the path. The output of the lexical chooser contains many features which are not relevant for syntactic realization. Therefore, extracting from the output of the lexical chooser just the features required by SURGE would increase efficiency significantly. Manual experimentation has shown that removing unnecessary embedding and cross-linking in the input to SURGE cuts the runtime of the realization module by a factor larger than 2.

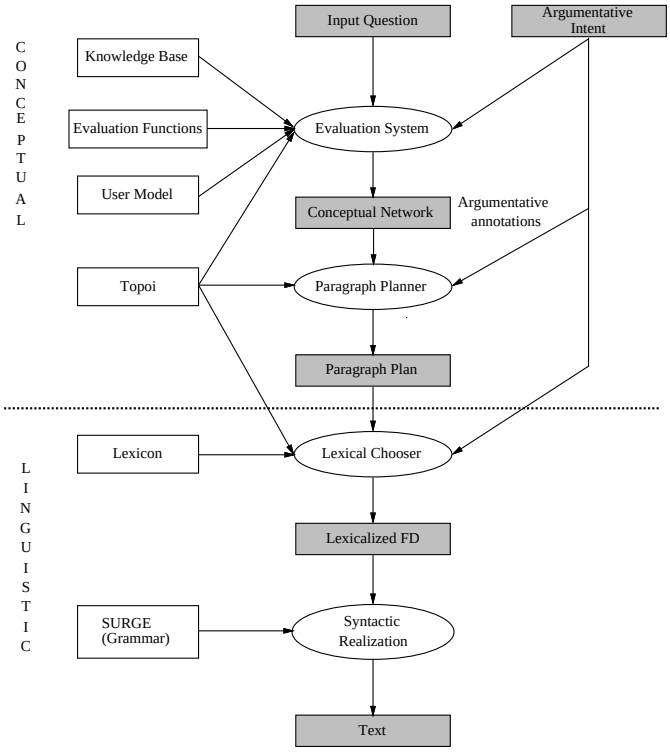


Figure 7-1: Architecture of ADVISOR II

The arguments chains become the input to the paragraph planner. The paragraph planner performs two functions: first it determines a final orientation for the evaluation of the class on the target scale, given the several individual scalar evaluations produced by the evaluation system. Second, it organizes the paragraph, choosing an order for the arguments, and putting each observation and evaluation as directive or subordinate branches of a hierarchical paragraph plan. The paragraph planner accesses the topoi base to perform its task. The paragraph planner is not the focus of this work. The current implementation is primitive and needs further work. This current implementation is described in Sect.7.5.

The paragraph plan produced by the planner is then sent to the lexical chooser. The operation of the lexical chooser and the structure of the lexicon are described in Chap.6. The lexical chooser is fully implemented in FUF. Its output is a lexicalized FD for the whole paragraph, where all open class words are specified, and all decisions concerning the linguistic structure have been made. Practical implementation issues of the lexical chooser are presented in Sect.7.6.

This lexicalized FD is finally fed to the FUF surface realization component which produces the text of the paragraph. This operation is entirely realized by calling the FUF unifier.

7.3. Knowledge Representation

Knowledge about the classes and the student is represented in a knowledge base implemented in the CLASSIC formalism [Resnick et al. 90; Brachman et al. 90], a KL-ONE type of language. Figure 7-2 shows a fragment of the class definitions I use in the ADVISOR II domain.[61] In the figure, a transversal arc connecting a set of edges indicates that the specializations form a disjoint partition of the parent class. As mentioned in Chap.6.2, this ontology is not motivated by a linguistic analysis. It is separated in two layers: a domain-independent upper-model, and a domain-specific specialization. The same upper-model is also being used in another domain ([Robin 92b] in the basketball domain) demonstrating its portability. The knowledge base is designed to encode all the information necessary for the evaluation functions to perform their tasks. The concepts and relations are defined to make this inferencing as easy to formulate as possible. Inheritance is used to reduce redundancy in the representation. Knowledge representation is critical to this application, because it limits what can be said about a class or a student.

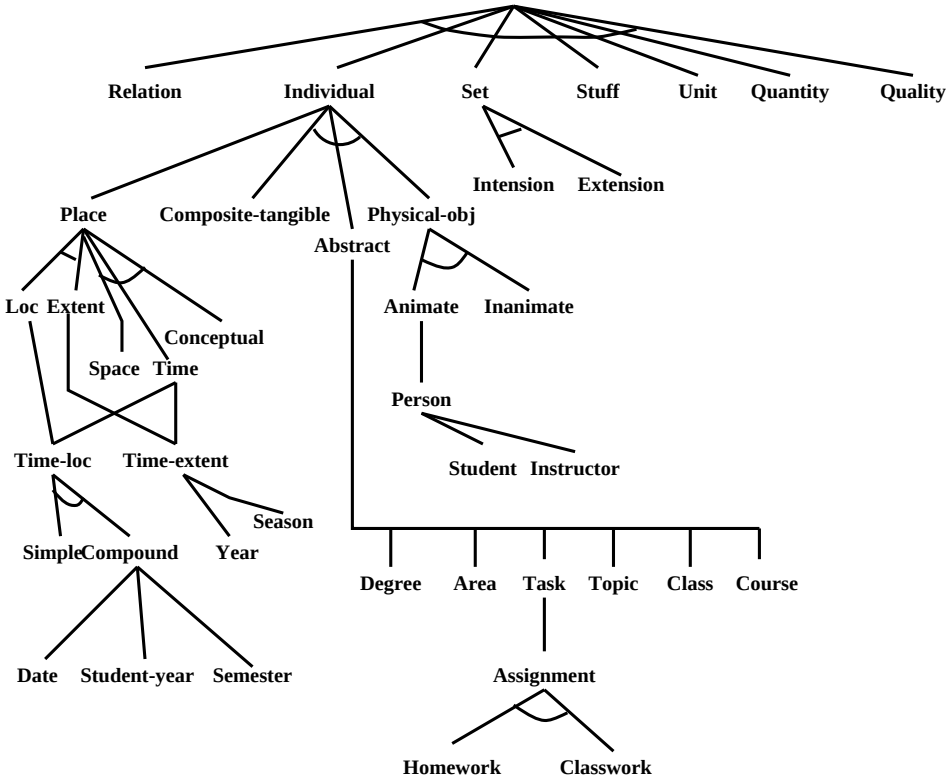


Figure 7-2: Fragment of CLASSIC knowledge base

Each of the generic classes shown in the figure can enter in relation with other entities. The main relations used in the ADVISOR II domain are shown in Fig.7-2. For each of these relations, a role filler class is listed. For example, a `class` object is related to a set of `task`s through the `assignments` relation.

The knowledge base contains a set of instances *i.e.*, specific individuals. Each instance is a member of one class

[61] I want to thank the CLASSIC team for making their software available. Most of the ontology has been designed by Jacques Robin. I am grateful for his assistance.

Roles for object `class`:

Roles	Role Fillers
area	Area
prerequisites	Class
normal-sequence	Student-year
topics	Topic
assignments	Task
followup	Class
required	Boolean

Roles for object `Course`:

Roles	Role Fillers
class	Class
instructor	Instructor
days	Time-loc
start-time	Time-loc
end-time	Time-loc
semester	Time-loc
year	Time-loc

Roles for object `Student`:

Roles	Role Fillers
points	Quantity
taken	Course-taken
taking	Class
experience	Topic
interest	Abstract
degree	Degree
degree-candidacy	Degree

Figure 7-3: Roles defined in the knowledge-base

object and the fillers of its roles are specified. For example, the `AI` instance is a member of the `Class` concept, and it has role fillers specifying the topics covered in AI, the assignments handed out in the class etc. Figure 7-4 shows examples of CLASSIC definitions for a concept, a relation and an instance. The concept `class` is a primitive specialization of the `abstract` concept, as indicated by the statement `(disj-prim abstract univ-obj 1)`. This means that every class is an abstract entity but the CLASSIC description of the concept is not a necessary and sufficient condition for being a *class*. The concept description lists the roles of `class`, and for each role specifies the type of the fillers. For example, the relation `area` links a class to a set of instances of the `area` concept. Note that CLASSIC does not authorize circular references in concept definitions, therefore *class* cannot be used in the definition of *class*, and instead of `(all followup class)` the definition contains `(all followup abstract)`. For the roles `area` and `required`, a value restriction `(at-most 1 R)` imposes that any instance of the concept class has at most one filler for the role.

The entity *ai-class* is defined as an instance of the `class` concept. The `fills` statement provides values for each

```
;; The concept class is a primitive specialization of abstract
(cl-define-concept 'class '(and (disj-prim abstract univ-obj 1)
                                (all area area)
                                (at-most 1 area)
                                (all normal-seq student-year)
                                (all prereq abstract)
                                (all required boolean)
                                (at-most 1 required)
                                (all topic topic)
                                (all assignment assignment)
                                (all followup abstract)))

;; The roles first-name and last-name are relations with at-most one
;; value. (The value restriction is indicated by the t argument).
(cl-define-roles '(first-name last-name) t)

;; The ai-class
(cl-create-ind 'ai-class
               '(and class
                     (fills area ai)
                     (fills prereq data-structures)
                     (fills normal-seq junior)
                     (fills topic ai expert-systems vision nlp
                            search robotics game-playing logic)
                     (fills required yes)
                     (fills followup vision nlp expert-system kr
                            robotics)
                     (fills assignment
                            prog1 prog2 prog3 prog4
                            paper1 project1)))
```

Figure 7-4: CLASSIC concept, relation and invidual definitions

role. For example, the value of the `normal-seq` role of `ai-class` is the instance `junior`, indicating that the AI class is taken in general in the junior year of college. Several instances can be listed for an attribute. For example the value of the role `followup` is a list of 5 courses.

The user model is encoded in CLASSIC. Each user is represented by an instance in CLASSIC. Figure 7-5 shows the user profile of a student named John Doe. John is studying towards his BS, is in his junior year in the Fall of 92. He has already taken 30 points in the program, and has taken the Computer Science courses *Introduction to Programming*, *Data-Structures* and *Discrete Math*. The grades he got in each one of these classes are also recorded in the knowledge base in the instances `intro11` etc which are instances of the `course-taken` concept. The `profile+` and `profile-` roles list topics and tasks in which the student has good or bad experience, and `interest` lists topics in which the student has expressed interest. This information is currently entered in that form in a text file before a session with ADVISOR II starts. No facility has been developed to enter the information interactively.

In summary, the knowledge base and user model are the main knowledge sources of the system. They are both encoded in the CLASSIC knowledge representation system, a standard KL-ONE type of formalism. The ontology developed for the application is not motivated by linguistic considerations, but uniquely by the type of inferences that are necessary in the domain. The ontology is split into a portable upper-model, and a domain specific layer, which specializes the upper-model. From the viewpoint of the generator, the knowledge base produces a simple network of individuals, sets and relations, which, as discussed in Chap.6, provides enough information to the lexical chooser to perform its task.

```
(cl-create-ind 'user1
               '(and student
                     (fills first-name "John")
                     (fills last-name "Doe")
                     (fills degree-candidacy bs)
                     (fills year junior)
                     (fills semester f92)
                     (fills points 30)
                     (fills taken intro11 data-structures11 discrete11)
                     (fills concentration ai)
                     (fills profile+ programming)
                     (fills profile- math paper1)
                     (fills interest ai nlp)))
```

Figure 7-5: A user profile

7.4. The Evaluation System

The evaluation system is the inferencing module of ADVISOR II. It takes as input a query, in the form of a triplet (class, scale, student-profile) and returns as output a list of argumentative evaluations of the class along the scale, tailored to the particular student. An example of evaluation is shown in Fig.7-8. The input query in this example is to evaluate the AI class on the scale `take` for a student described by the profile `student1`. This student profile indicates among other things that the student is interested in NLP, has little experience writing papers, dislikes theory, and enjoys programming.

As discussed in Chap.2, the output of the evaluation system is a list of argument chains. Each chain is made up of three elements: a knowledge base observation, a judgment and a chain of topoi. The observation is a conceptual network, which is extracted from the knowledge base and the user model. It contains facts about the class and the user that support a first argumentative evaluation, the judgment. The judgment is the position of the class on a scale. The topoi chain links this first evaluation to the target scale, by chaining through topoi. For example, in Fig.7-8, the AI class is first evaluated as high on the scale of workload based on the knowledge base observation graphically depicted in Fig.7-9. This conceptual network represents the fact that AI has a set of assignments consisting of writing papers, and the student has little experience writing papers. As a consequence, AI is judged as potentially requiring a high work load from this student. This first evaluation is performed by an *evaluation function*. The list of evaluation functions implemented in ADVISOR II is shown in Fig.7-6. The notation in this figure is to use path to access the value of roles in the knowledge base. For example, the term `class.assignments.number` accesses the number of fillers of the role `assignments` for the instance `class`. The notation `class.[assignments/programming].number` denotes the number of assignments of `class` which are instances of the `programming` concept; that is, the number of programming assignments in `class`.

Evaluation functions are implemented as COMMON LISP functions. Figure 7-7 shows the code of the rule EVAL-12 which triggered the evaluation of the first observation shown in Fig.7-8. All evaluation functions are straight translations of the notation shown in Fig.7-6 and have the same structure. On top of the CLASSIC query functions (whose name starts with the prefix `cl-`) a collection of knowledge-base access functions has been developed to query the knowledge-base and extract the desired information to build the FD representation of the conceptual network. Among these functions, the function `get-role-fillers` is used to return the list of role-fillers of an input list of instances for a given relation (the notation `@r{rel}` denotes the CLASSIC relation named `rel`). The COMMON LISP function contains two parts: in the let-binding, a query is applied to the knowledge base, and the result is bound to LISP variables; in the second part, an FD is built, if the query is successful, which has the form: `((left <evaluation>) (justification <conceptual-network>))`. This FD can be seen either as the left-hand side of a topos, or as a simple argumentative evaluation, rooted in an observation. In the justification part of the FD, the value of each slot is computed by querying information from the knowledge base. Two COMMON LISP functions

DOMAIN-PROGRAMMING
If (user.programming -) ∧ (class.[assignments/programming].number >0) → (programming +)
If (user.programming given) ∧ (class.[assignments/programming].number = 0) → (programming -)
If (user.programming +) ∧ (class.[assignments/programming].number >3) → (programming +)
If (user.programming +) ∧ (class.[assignments/programming].number <= 3 > 0) → (programming none)

DOMAIN-MATH
If (user.math -) ∧ (class.area = theory) → (mathematical +)
If (user.math -) ∧ (Y ∈ class.topic) ∧ (Y.area = theory) → (mathematical +)
If (user.math given) ∧ (class.area ≠ theory) → (mathematical -)

DIFFICULTY
If (Y ∈ user.experience) ∧ (Y ∈ class.topic) → (difficulty -)

INTEREST
If (Y ∈ user.interest) ∧ (Y ∈ class.topic) → (interest +)

WORKLOAD
If (class.assignments.number > 4) → (workload +)
If (class.assignments.number < 2) → (workload -)
If (Y ∈ user.profile-) ∧ (X ∈ class.assignments) ∧ (X.hw-type = Y) → (workload +)

LEVEL
If (user.year = Y) ∧ (class.normal-seq = Y) → (level -)
If (user.year = Y) ∧ (class.normal-seq = X) ∧ (X > Y) → (level +)

Figure 7-6: List of evaluation functions

are used for this purpose: `cl-ind-name` is a CLASSIC function which returns the name of an instance. It is used to build FD descriptions of individuals. `Fd-make-set` is a function I have implemented to format a list of instances as a set, using the set representation discussed in Chap.6. If the knowledge-base query does not succeed, the evaluation function returns `*fail*`.

Evaluation functions perform a critical task in the system: they bridge the gap between objective knowledge and judgment. They are also the component of the system which contains most of the bias of the developer, and they are the least portable from domain to domain.

Evaluation functions also provide the system with the semantics of the scales manipulated later. The eight scales identified in the domain are:

- Goodness
- Interest
- Importance
- Level
- Difficulty
- Workload
- Programming
- Mathematical

Several of these scales are non-linear, in the sense defined in Sect.6.7.2 (p.199). For example, a course can be difficult because it has a high workload, or because it covers a topic with which the student has little experience. Two independent criteria can thus be used to reach the same conclusion.

```
(defun EVAL-12 (class student)
  ;; If (Y in user.profile-) &
  ;;    (X in class.assignments) & (X.hw-type = Y) =>
  ;;    (workload +)

  ;; Query knowledge-base and extract useful information
  (let* ((profile- (cl-ind-role-fillers student @r{profile-}))
         (a1 (cl-ind-role-fillers class @r{assignment}))  ;; X
         (t1 (get-role-fillers a1 @r{hw-type}))           ;; Y
         (inter (intersection profile- t1)))              ;; X k Y #W# j
    (If inter
        '(
          ;; Judgment
          (left ((evaluated {^2 justification class})
                 (scale ((name "workload")))
                 (orientation +)))

          ;; Observation (conceptual network)
          (justification
           ((rule EVAL-12)
            (assignments ,(fd-make-set inter 'assignment
                                       '(realize-extension no)))
            (class ((cat class) (name ,(cl-ind-name class))))
            (hw-activities ,(fd-make-set inter 'hw-activity
                                         '(realize-extension yes)))
            (class-relation ((name assignments)
                             (1 {^2 class})
                             (2 {^2 assignments})))
            (user-relation ((name experience)
                            (orientation -)
                            (1 ((cat student) (name hearer)))
                            (2 {^2 assignments})))
            (object-relation ((name hw-type)
                              (1 {^2 assignments})
                              (2 {^2 hw-activities})))))))
        *fail*)))
```

Figure 7-7: Code for an evaluation function

The last element of an argument chain, after the observation and the judgment, is the topoi chain. The topoi chain links the judgment, which is a scalar evaluation, to an evaluation on the target scale. For example, in the first evaluation in Fig.7-9, the evaluation function produced a judgment on the workload scale. The target scale was `take` (a synonym of goodness in the domain). So a topoi chain links `workload` to `take`: /+ workload, + difficult/ and /+ difficult, -take/. Only the evaluation functions which can be related to the target scale are kept in the output of the evaluation system.

The topoi used in ADVISOR II are listed in Fig.7-10. Certain of these topoi only apply for a certain class of users. For example, the topos /+ programming, + interest/ is only triggered for students who have already expressed interest in programming (the ‘‘hackers’’). Such topoi are related to stereotypes in the user model. Topoi are represented in a grammar in the FUF formalism, which also encodes chaining. Chaining is implemented in FUF using a grammar very similar to the `append` example listed in Sect.3.4.1 (p.74). In addition to chaining, the topoi grammar implements simple equivalence manipulations between topoi, for example, /+L, +R/ is equivalent to /-L, -R/. All the rules defined in the grammar are listed in Fig.7-11 in a Prolog-like notation (more concise than the FUF notation).

Like all FUF programs, the topoi grammar is bidirectional. In this case, it means that the grammar can be used to check whether an FD `((left X) (right Y))` is known to the system when `X` and `Y` are known, or it can be used in a productive manner, to find all the topoi or chains of topoi starting with a given left-hand side and/or ending in a given right-hand side. This is the mode in which it is used to link each evaluation produced by the evaluation

```
Query: (AI, TAKE, STUDENT1)

(1) KB Observations:    topics-of(ai, topics1),
                        member-of(topics1, logic),
                        area(logic, theory),
                        user-profile(experience, theory, -)
Judgment:               <theory(AI), +>
Argumentative Linking:  /+ theory(AI), + difficult(AI)/
                        /+ difficult(AI), - take(student, AI)/

(2) KB Observations:    assignments-of(ai, assignments1)
                        member-of(assignments1, a1),
                        activity-of(a1, paper-writing),
                        user-profile(experience, paper-writing, -)
Judgment:               <workload(AI), +>
Argumentative Linking:  /+ workload(AI), + difficult(AI)/
                        /+ difficult(AI), - take(student, AI)/

(3) KB Observations:    assignments-of(ai, assignments1),
                        subset-of(assignments1, prog-assignments,
                                        activity(X, programming)),
                        cardinal(prog-assignments) > 2,
                        user-profile(interest, programming, +)
Judgment:               <programming(AI), +>
Argumentative Linking:  /+ programming(AI), + interest(AI)/
                        /+ interest(AI), + take(student, AI)/

(4) KB Observations:    topics-of(ai, topics1),
                        member-of(topics1, nlp),
                        user-profile(interest, nlp, +)
Judgment:               <interest(AI), +>
Argumentative Linking:  /+ interest(AI), + take(student, AI)/
```

Figure 7-8: Argumentative chains produced by the evaluation system

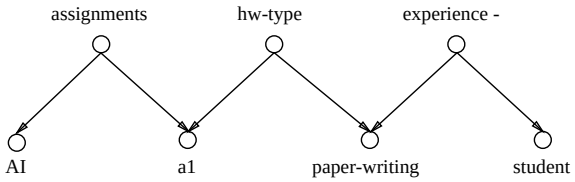


Figure 7-9: A knowledge-base observation

functions to the target scale. The topoi grammar is separated in two modules: a first disjunction lists all the canonical topoi in the domain (those listed in Fig.7-10) and a second module implements the transformations shown in Fig.7-11. This second module is domain-independent, and can work with any set of canonical topoi.

At the end of the chaining stage, the evaluation system has produced a number of argument chains, each containing an observation, a judgment and a topoi chain ending with the target scale, with a valuation of either + or -. Figure 7-12 shows the FD form of a chain. In the example in Fig.7-9, four such chains are produced, two ending with a +take evaluation, and two ending with a -take. So the evaluation system does not quite produce an answer to the question. What is missing is a module to compose these chains and perform the trade-off analysis to eventually choose an answer to the query: +take or -take, given the 4 chains. This trade-off analysis is not performed by the evaluation system but by the paragraph planner, as discussed in the next section.

In summary, the evaluation system is characterized by the following properties:

+ workload / + difficulty
+ workload / + time-required
+ difficulty / + workload
+ difficulty / + time-required
+ difficulty / - take
+ time-required / + workload
+ programming / + time-required
+ programming / + interest [User-Model]
+ math / + difficulty [User-Model]
+ math / + interest [User-Model]
+ level / + difficulty
+ interest / + take
+ importance / + take

Figure 7-10: List of topoi used in ADVISOR II

;; Tautology, the more X, the more X is a valid topos.
topos(S X, S X).

;; Sign equivalence
topos(+ X, + Y) ← topos(- X, - Y).
topos(+ X, - Y) ← topos(- X, + Y).
topos(- X, + Y) ← topos(+ X, - Y).
topos(- X, - Y) ← topos(+ X, + Y).

;; Chaining
topos(S1 L, S2 R) ← topos(S1 L, S M) ∧ topos(S M, S2 R).

Figure 7-11: Rules implemented in the topoi grammar

- The input to the evaluation system is a triple (class, scale, student-profile).
- A set of evaluation functions serve three purposes: they bridge the gap between objective facts and user-dependent judgments, they extract relevant information from the knowledge base and they convert from the formalism of the knowledge base to the FD formalism of the generator.
- A topoi base is used to chain from judgments to target scales.
- The output of the evaluation system is a list of argument chains, each comprising a knowledge-base observation, a judgment and a topoi chain ending with the target scale. The output is not ‘‘resolved’’ in the sense that both + and - evaluations on the target scales are kept.

The portable (domain independent) elements of this system are:

- An application-programming interface (API) to CLASSIC extending the CLASSIC native query API with list manipulation functions and functions converting CLASSIC objects to the FD notation used by the lexical chooser.
- A generic topoi manipulation grammar written in FUF which, once a set of canonical topoi are defined, provides the system with a procedure to check that two scales can be related by an argumentative chain.

The system assumes that three pieces of domain-specific knowledge are defined in input:

- A set of scales.
- A set of canonical topoi.
- A set of evaluation functions.

```
;; Many topics #L# + Interest #L# + Take
(def-test chain
  "AI covers two interesting topics.  It should be a good class."
  ((ao ((cat topos)
        (left
         ((evaluated {justification topics})
          (scale ((name "cardinality")))
          (orientation +)
          (type composite)
          (value {right})))
        (chain1 ((cat topos)
                 (left {ao left})
                 (right
                  ((evaluated {justification class})
                   (realization-mode clause)
                   (scale ((name "interest")))
                   (orientation +)
                   (value +)))))
        (chain2 ((cat topos)
                 (left {ao chain1 right})
                 (right {ao right})))
        (right ((cat evaluation)
                (evaluated {justification class})
                (scale ((name "take")))
                (orientation +)))))
   (justification
    ((rule eval-9)
     (topics
      ((cat set) (index top1) (kind ((cat topic))) (cardinality 2)
       (extension
        ((car ((semr ((cat topic) (name ai)))))
         (cdr ((car ((semr ((cat topic) (name nlp))))) (cdr none)))))))
     (class ((cat class) (index ai1) (name ai-class)))
     (class-relation
      ((name topics)
       (1 ((semr {justification class})))
       (2 ((semr {justification topics})))))
     (user-relation
      ((name interest) (orientation +)
       (1 ((semr ((cat student) (name hearer)))))
       (2 ((semr {justification topics}))))))))))
```

Figure 7-12: FD format of an argument chain

Scales and topoi can be identified in the domain by investigating a corpus of naturally occurring conversations (an inductive analysis) or by performing deductive experiments. I have discussed in Sect.6.7.1 how I systematically listed all adjectives modifying a class in the corpus. Adjectives are scalar in nature and their occurrence indicates which scales are most often mentioned in the domain. An analysis of the argumentative connectives (*so*, *but* etc.) is similarly useful to identify the topoi used in the domain. I have discussed in Sect.2.3.2 Osgood's semantic differential, and its potential use for empirically verifying the validity of the scales identified in the domain. In a practical application, setting up an experiment similar to the semantic differential with potential users would likely be very beneficial.

In [Hirschberg 85, Chap.7.2.4], Hirschberg presents a system deriving scalar information from a knowledge base in a mostly domain-independent manner. For example, when dealing with sets, inclusion defines a partial order over the set of subsets which can serve at the basis of scalar implicature. Similarly, an *isa* hierarchy define a partial ordering over the concepts of a terminological knowledge base. Using such scales would be useful for identifying the ''natural'' scales which serve most often as the basis of composite evaluations, and, therefore, would be useful to identify evaluation functions. The only such natural domain-independent scale used in ADVISOR II is the scale of cardinality. But this domain-independent approach would not help in identifying the domain-specific scales I have just discussed. Knowledge of these scales is part of the domain expertise, and cannot be uncovered by domain independent techniques.

7.5. Paragraph Planning

The paragraph planner is the least developed component of the current ADVISOR II prototype. The current implementation of the paragraph planner requires the assistance of the user at most decision points. In a practical system, expert modules would make these decisions based on information about the user and on a sort of ‘‘system model’’ - a data-structure encoding the biases and goals of the system. In this section I identify the decision points in the paragraph planners, and for each one, list possible outcomes. I then describe the output of the paragraph planner, and how it is built from the list of argument chains provided by the evaluation system.

The main issues that the paragraph planner must decide can be classed along two dimensions: selection and aggregation. Selection means that the planner must select a subset of the information provided by the evaluation system for inclusion in the paragraph. Aggregation means that the structure of the paragraph must be constructed.

Several options are possible for selection: one might always convey all the propositions of the input in the answer, or select only a subset of those. When selecting a subset, different criteria can be used: select a subset of a fixed size (*e.g.*, only one proposition, probably the ‘‘best’’ according to some metric), select only the arguments that support the argumentative intent of the answer, select only ‘‘relevant’’ arguments, if an operational definition of relevance can be used. Even when the decision is to include all propositions in the answer, one still has to decide whether to leave out propositions that can be easily recovered or inferred by the hearer.

Part of the selection problem is determining which orientation the answer should have eventually. This problem requires performing a trade-off analysis between the arguments provided by the evaluation system. For example, with the input shown in Fig.7-9, the main trade-off to be considered is whether an interesting course should be taken even though it is difficult, or vice-versa whether a difficult course should be avoided even though it could be interesting. I have not implemented a system to rank arguments in this manner. Instead, ADVISOR II asks a human assistant[62] for the eventual orientation of the answer, and structures the paragraph accordingly.

A second aspect of the selection problem is to determine how many chains from the input should be combined in the paragraph. ADVISOR II currently limits the paragraph to a combination of two chains. Since each chain is limited to a maximum of three relations plus two evaluations, the maximum number of clauses in a generated paragraph is 10. This limitation is arbitrary and has only been adopted for ease of implementation.

Finally the last aspect of the selection problem, is to actually select the chains to be realized in the paragraph. ADVISOR II requires the human assistant to select two chains from the evaluation output manually. For the example shown in Fig.7-9, the selection session is shown in Fig.7-13. The selection can also be made randomly. The result of the planning with different selections is shown in Fig.7-14.

After selection, the second dimension characterizing paragraph planning concerns aggregation and structure. The problem is to build a discourse structure which covers the selected information. This is performed in two steps: merging and hierarchy formation.

The first step is merging. Recall that the output of the evaluation system (cf. Fig.7-9 p.220) consists of argument chains. The paragraph planner can factor out common parts of several chains. For example, chains 1 and 2 both end with the same links /+difficult,-take/. It is, therefore, possible to build a tree-like structure, as shown in Fig.7-15. In this tree, the leaves correspond to the observations found by the evaluation functions, that is, to propositions extracted from the knowledge base.

Once merging is performed, the second step is to construct a discourse structure capturing the hierarchy imposed by the choice between directive and subordinate act. One example is shown in Fig.7-16, which shows the two steps in the structuring of the paragraph corresponding to chains 1 and 2 in Fig.7-14. First, the common parts of the chains

[62]Since ADVISOR II is not fully automatic, it requires assistance from a sort of human ‘‘oracle’’ besides the end-user. I refer to the person playing this role as the assistant.

Advisor> Please select 2 chains out of the following,
starting with the conclusion you want to support:

1: + theory --> + difficult --> - take
2: + workload --> + difficult --> - take
3: + programming --> + interesting --> + take
4: + interesting --> + take

Enter two numbers: 1, 3

Output Paragraph:

AI has many programming assignments, so it should be interesting.
But it covers logic, a very theoretical topic, so it could be difficult.
I would not recommend it.

Figure 7-13: Interactive selection in paragraph planning

are merged. Then a tree-like data-structure is built, in FD format, using the labels introduced in Sect.6.8.5. Three labels are used: directive, subordinate and AO (for argumentative orientation). The directive segment expresses the ''point'' of the whole discourse segment. The subordinate segment brings support or elaboration for this point. The AO of a segment captures the argumentative evaluation that the segment expresses. Both directive and subordinate can be embedded discourse segments with their own decomposition, or, eventually, simple utterances corresponding to the expression of an observation with a certain AO.

At the leaves of the discourse structure, when an utterance is formed, the labels are PC (for propositional content) and AO. The lexical chooser can then decide to realize this propositional content in a single clause or in a larger discourse segment, with its own directive and subordinate structure. This second level structuring is performed by the lexical chooser only, and the representation sent by the paragraph planner is not committed to one form (clause) or the other (complex discourse segment). This reflects the fact that the lexical chooser considers the AO as a floating constraint, and can realize it within the clause, or by adding a connected clause. For example, in *AI has many programming assignments, so it should be interesting*, the evaluation `+ interest` is realized by a clause, whereas in *AI requires many theoretical topics*, the evaluation of `+ difficulty` is realized by the choice of the verb *require* which carries a connotation. So the paragraph planner does not decide in advance how the AO will be realized.

In contrast, the paragraph planner determines that the AO of the chains further along the chain from the justification are to be realized by separate clauses.

A slightly different form of merging is performed in Fig.7-17, where the nodes `+take` and `-take` are merged, even though their signs are different, to create a contrastive relation between the corresponding segments in the text (realized by the connective *but*). Figure 7-18 shows the output of the planner when combining chains 1 and 2 in the FD format sent to the lexical chooser. The original justification and AO constituents of each chain appear under the features justification1 and justification2, and AO1 and AO2. The top part of the FD contains the paragraph structure under the features directive and subordinate. The directive move is a simple segment, only realizing a single AO. The subordinate move is recursively a complex discourse segment.

In summary, the paragraph planner relies on two types of decisions:

- Selection of a subset of the chains provided by the evaluation system.
- Structuring of the selection by distinguishing between directive and subordinate elements.

In its current implementation, the planner uses stub functions to perform these decisions. The stub functions either query the user or perform the decisions randomly. Once these decisions are performed, the planner produces a discourse structure description in a format appropriate for the lexical chooser. A further simplifying assumption of

Observation1: + theory → + difficult → - take
Observation2: + workload → + difficult → - take
AI covers logic, a very theoretical topic
and it has many assignments which consist of writing papers.
You have little experience writing papers.
Therefore it could be quite difficult.
I would not recommend it.

Observation1: + theory → + difficult → - take
Observation3: + programming → + interesting → + take
AI has many programming assignments, so it should be interesting,
but it covers logic, a very theoretical topic, so it could be difficult.
I would not recommend it.

Observation1: + theory → + difficult → - take
Observation4: + interesting → + take
AI covers many interesting topics, such as NLP, vision and KR.
But it deals with logic, a very theoretical topic, so it could be
difficult. I would not recommend it.

Observation2: + workload → + difficult → - take
Observation3: + programming → + interesting → + take
AI should be interesting because it has many programming assignments.
But it has many assignments which consist of writing papers.
You have little experience writing papers.
So it could be very difficult.
I would not recommend it.

Observation2: + workload → + difficult → - take
Observation4: + interesting → + take
AI deals with many interesting topics, such as NLP, Vision and KR.
But it has many assignments which consist of writing papers.
You have little experience writing papers.
So it could be difficult.
I would not recommend it.

Observation3: + programming → + interesting → + take
Observation4: + interesting → + take
AI has many programming homeworks and it covers a lot of interesting
topics such as NLP, Vision and KR, so it should be quite interesting.
I would recommend it.

Observation3: + programming → + interesting → + take
Observation1: + theory → + difficult → - take
AI covers logic, a very theoretical topic, so it could be difficult.
But it has many interesting topics such as NLP, vision and KR.
It should be a good class..

Figure 7-14: Paragraphs with different structures generated by ADVISOR II

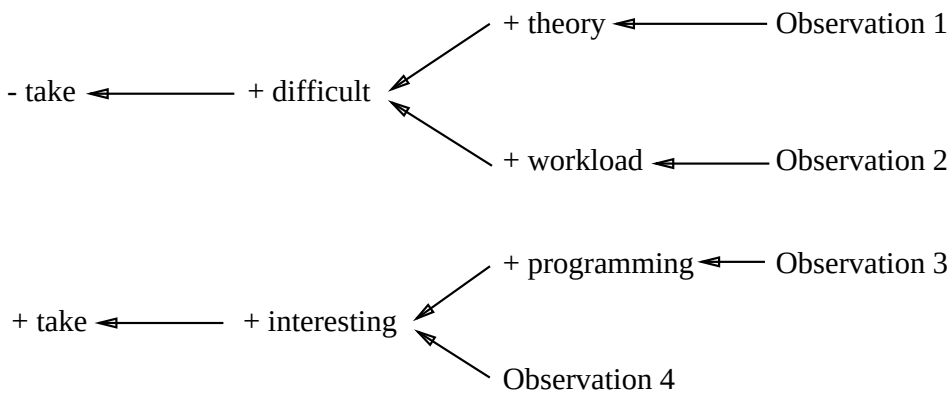
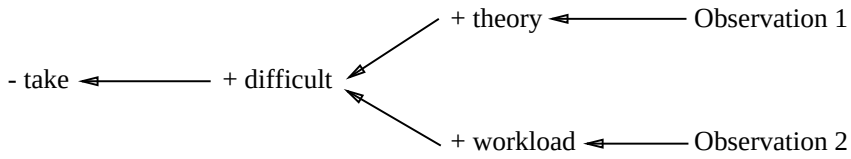
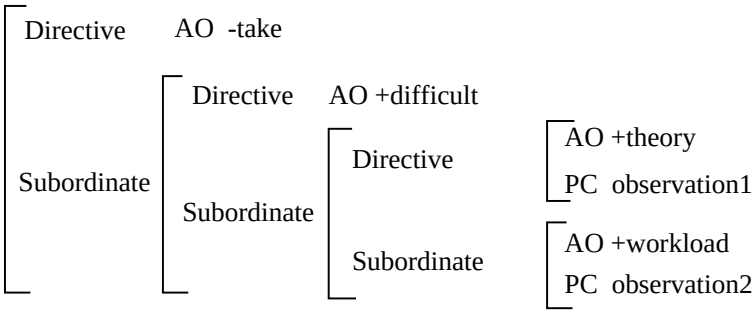


Figure 7-15: Factoring in content organization

Merging



Hierarchy Formation



Text

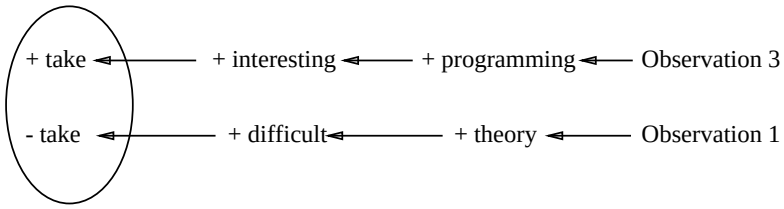
AI covers logic, a very theoretical topic Observation1
and it has many assignments which consist of writing papers. Observation2
You have little experience writing papers.
Therefore it could be difficult.
I would not recommend it.

Figure 7-16: Steps in planning the paragraph for selection (1,2)

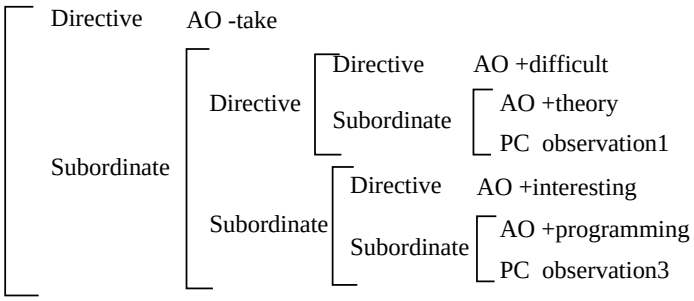
the implementation is that the planner selects at most 2 chains out of the output of the evaluation system for inclusion in the answer.

To replace these stub functions, a practical system would need to:

Merging



Hierarchy Formation



Text

AI has many programming assignments, so it should be interesting.
But it covers logic, a very theoretical topic, so it could be difficult.
I would not recommend it.

Figure 7-17: Steps in planning the paragraph for selection (1,3)

- Analyze the trade-offs involved in choosing between a + and a - orientation on the same scale when two chains of arguments support + and -.
- Rank the chains of arguments in terms of strength and persuasion potential, to select only the more efficient ones.

An innovative aspect of this planner is that some structuring decisions are left to the lexical chooser. Thus, the lexical chooser can decide to expand a segment into a complex discourse segment, or when possible, can realize it as a single clause. This architecture removes the burden from the paragraph planner of knowing which conceptual element can be realized by a single clause, and which ones require more than a clause. Specifically, the lexical chooser can produce a clause complex in two situations: when a floating constraints like the argumentative orientation does not find a site in the clause to be realized, and when the conceptual network is so complex that realizing it as a single clause would require deeply embedded modifiers.

```
(def-test plan13
  "AI has many programming assignments, so it should be interesting.
   But it covers logic, a very theoretical topic.
   So it could be difficult.  I would not recommend it."
  ((cat ds)
   (directive ((ao ((cat evaluation) (evaluated {justification class})
                    (scale ((name "take"))) (orientation -)))))
   (subordinate
    ((directive
      ((directive ((ao ((evaluated {justification class})
                        (scale ((name "difficult")))(orientation +)))))
       (subordinate ((ao ((cat topos) (left {ao1 left})
                          (right ((evaluated {justification class})
                                  (scale ((name "math")))
                                  (orientation +)))))
                     (pc {justification1})))))
     (subordinate
      ((directive ((ao ((evaluated {justification class})
                        (scale ((name "interest")))
                        (orientation +) (um yes) (value +)))))
       (subordinate ((ao ((cat topos) (left {ao2 left})
                          (right ((evaluated {justification class})
                                  (scale ((name "programming")))
                                  (value +) (orientation +)))))
                     (pc {justification2})))))))
   (ao1 ((left ((evaluated {justification1 topics})
                (scale ((name "cardinal")))
                (value {subordinate directive subordinate ao right})
                (type composite) (orientation +)))))
   (ao2 ((left ((evaluated {justification2 assignments})
                (scale ((name "cardinal")))
                (value {subordinate subordinate subordinate ao right})
                (orientation +) (type composite)))))
   (justification1
    ((rule eval-6)
     (topics ((cat set) (index top2) (kind ((cat topic)))
              (cardinality 1)
              (extension ~( ((semr ((cat topic) (name logic)))) ))))
     (area ((cat area) (index ar1) (name theory)))
     (class ((cat class) (index ai1) (name ai-class)))
     (class-relation ((name topics)
                      (1 ((semr {justification1 class})))
                      (2 ((semr {justification1 topics})))))
     (object-relation ((name area)
                       (1 ((semr {justification1 topics})))
                       (2 ((semr {justification1 area})))))
     (user-relation ((name experience) (orientation -)
                     (1 ((semr ((cat student) (name hearer)))))
                     (2 ((semr {justification1 area})))))))
   (justification2
    ((rule eval-4)
     (course ((cat class) (index ai1) (name ai-class)))
     (assignments ((cat set) (index hw3) (kind ((cat assignment)))
                   (cardinality 2)))
     (hw-activity ((cat hw-activity) (index hwa2) (name programming)))
     (class-relation ((name assignments)
                      (1 ((semr {justification course})))
                      (2 ((semr {justification assignments})))))
     (user-relation ((name experience) (orientation +)
                     (1 ((semr ((cat student) (name hearer)))))
                     (2 ((semr {justification hw-activity})))))
     (object-relation ((name hw-type)
                       (1 ((semr {justification assignments})))
                       (2 ((semr {justification hw-activity}))))))))))
```

Figure 7-18: Output of the paragraph planner

7.6. Lexical Choice and Surface Realization

The operation of the lexical chooser was described in detail in Chap.6. I discuss in this section the following implementation issues which were not mentioned there:

- Realization of purely argumentative segments.
- Dealing with the floating nature of the argumentative orientation.
- Choice of a layered representation.
- Pronominalization.

7.6.1. Realization of Purely Argumentative Segments

In Chap.6, I assumed that the input to the lexical chooser contains a conceptual network annotated by argumentative evaluation features. In the input shown in Fig.7-18, however, several segments contain only an argumentative orientation (AO). In this case, the lexical chooser realizes the argumentative evaluation as a clause, using the mapping from scale to clause discussed in Sect.6.7.3. For example, the lexical entry for the scale `take` specifies that the clause patterns *I recommend* <*evaluated*> and <*evaluated*> *is a good* <*kind*> are appropriate. Since these patterns are represented as complete FDs and not simply as templates, the lexical chooser can still add modality as appropriate, negation and adjective intensifiers, and perform pronominalization, producing a variety of clauses like *I would not recommend it, it would be a very good course, AI is a good course* etc.

7.6.2. Dealing with Floating Constraints

The argumentative orientation (AO) feature can be lexicalized at different levels of the linguistic structure, depending on which lexical resources are available. The implementation of this decision in the lexical chooser makes use of the `bk-class` and `wait` features, in a way very similar to the method shown in Sect.4.1.3.

The AO feature is a floating constraint when the input contains both an AO and a propositional content (PC) feature (when only the AO is present, it is mapped to a clause, as explained above). When an input containing both a PC and an AO is processed, the following steps are performed:

- PC is mapped to a linguistic structure: PC contains a conceptual network, so the steps in mapping it to a linguistic structure, as explained in Sect.6.4.6, are to select a perspective on the network (the dominant relation) and map attached relations to either arguments of the clause or to independent clauses, if no modifier slot is available in the main clause.
- The AO is then mapped onto a linguistic constituent: this is the step where the floating nature of the AO constraint plays a role.

The AO mapping is difficult because:

- It must be performed from the top level discourse segment constituent, since it can be mapped to any location within this segment.
- The attachment of the AO constraint at a given level constrains the lexicalization of the target constituent. For example, if AO is attached at the clause level, a different main verb is selected than if AO is attached at the NP level.
- The decision whether the AO constraint can be mapped at a given level cannot be made until the constituent at this level is lexicalized. For example, the AO can only be attached at the clause level if a verb with the appropriate connotation is available in the lexicon.
- The range of all sites where the AO can be attached is not known until all the linguistic structure has been built. But the linguistic structure is built incrementally, as each level is lexicalized. Therefore, there cannot be a single site in the lexical chooser which tries all possible attachment sites for AO in turn.

The main idea of the implementation is to identify sites of attachment for the AO features at each level, and to let each lexical head percolate the AO feature down to the appropriate subconstituent. This approach is necessary because the linguistic structure emerges dynamically as each lexical head is realized.

Specifically, at the top level the AO feature can be mapped in three different ways, depending on the type of argumentative evaluation:

- If the AO evaluation is absolute or relative, it can be realized by the NP being the focus of the evaluation, by forcing a cardinal determiner or a comparative determiner to be used (*e.g.*, *AI has 6 assignments* or *AI has many/more assignments*).
- If the AO evaluation is composite, then two evaluations need to be mapped onto the linguistic structure: left (in the domain, left is always an evaluation on the cardinality scale, which is the only objectively measurable quantity) and right. Left can be mapped onto the determiner of the evaluated constituent (*e.g.*, *AI has many assignments*), or as a non-restrictive modifier of the NP (*e.g.*, *AI has 6 assignments, which is a lot*). Right can be mapped either to the top level clause, to the evaluated constituent, or to a connected attributive clause.

When an AO evaluation is attached to an NP, similarly, several sites can be selected to realize it within the NP structure:

- The determiner sequence can realize evaluations on the cardinality scale (*e.g.*, *many assignments*).
- The describer can be filled by a scalar adjective (*e.g.*, *interesting assignments*).
- The classifier can be filled by a scalar noun or adjective (*e.g.*, *theoretical topics*).
- The qualifier can be an attributive clause realizing the AO (*e.g.*, *a class which require a lot of work*).

These attachment sites are placed in order of priority in the grammar for NPs and for clauses, so for example, choosing a verb with a connotation is preferred over adding a connected clause, and adding a describer is preferred over adding a relative clause. If, however, no adjective can be found in the lexicon, then the lexical chooser resorts to using a relative clause to realize the AO constraint.

The overall strategy to map the AO is:

1. Attach the AO at the current level
2. Try to realize the current constituent with the AO
3. If no lexical head can be found that satisfies the AO, map the AO to one of the governed constituents in the order of preference (this may require adding a new constituent specifically to realize the AO).

As is always the case with floating constraints, this strategy can trigger expansive backtracking, because, for example, it takes a long time to realize the the AO cannot be realized at the NP level. In order to handle the AO mapping efficiently, bk-class and wait annotations are added to each disjunction in the lexical chooser implementing the AO mapping. As a consequence, lexical choice is performed fairly quickly on the typical networks processed in ADVISOR II: the average number of backtracking points used for lexical choice is 100 to 150.

7.6.3. Choice of a Layered Representation

A second implementation point of the lexical chooser is the choice of a layered representation. A layered representation is required because the lexical chooser maps a conceptual representation to a linguistic structure, so both representations need to be represented in an FD. There are three options to encode such a dual-representation in an FD:

- The two structures are represented as two disjoint features as in `((linguistic X) (conceptual Y))`.
- The conceptual structure is embedded under the linguistic constituent, and each linguistic constituent contains an embedded feature describing the conceptual elements it realizes. The FD looks like `((cat clause) (semr Y) ...)` (`semr` stands for semantic representation). This representation makes it easy to follow the relation from linguistic to realized conceptual element.

- The linguistic structure is embedded under the conceptual constituent, and each linguistic constituent is embedded under the conceptual element it realizes. The FD looks like `((cat relation) (realization X) ...)`. This representation makes it easy to follow the relation from conceptual element to linguistic realization.

All three representations make the implementation possible, but the second one was selected for the convenience it provides. With the first approach, all paths establishing a link between linguistic and conceptual structures must pass through the root of the total FD. This poses a problem to represent the mapping between conceptual element and linguistic realization in the lexicon. Consider the following example:

```
((conceptual ((cat relation)
              (1 ((cat class) (name AI))) ...))
 (linguistic ((cat clause)
              (participants ((carrier #))))))
```

The lexicon must be accessed to find out how the individual `((cat class) (name AI))` is realized. The lexicon establishes the mapping as in:

```
((conceptual ((cat class) (name AI)))
 (linguistic ((cat proper)
              (lex "Introduction to Artificial Intelligence"))))
```

But this does not allow the filling of the slot marked # in the total FD, because the conceptual and the linguistic constituents are far apart in the total FD (one is at address `{conceptual 1}`, the other at address `{linguistic participants carrier}`). To actually fill the slot # with the value returned by the lexicon, the lexical chooser would need to first record the linguistic realization under the conceptual feature, and then establish a path pointer to the # position. The end result is that the conceptual feature actually contains all the linguistic material, which defeats the purposes of separating them. In addition, this scheme requires the placement of a long path between every pair conceptual element / linguistic realization. Tracing such long paths is the major source of inefficiency in the current implementation of the FUF unifier. So this representation can be inefficient. Finally, maintaining these paths is not easy to implement using the natural ‘‘top down’’ control regime of FUF, since it requires the traversal of two structures concurrently, whereas FUF is good at traversing only one structure at a time.

Since whenever an embedded conceptual element is looked up in the lexicon, the lexicon returns a pair concept / realization, it is more efficient to represent the two elements together. Therefore, only options 2 and 3 remain: linguistic on top or conceptual on top. I decided to have the linguistic element on top because, in this manner, the output of the lexical choice is a linguistic structure, which just happens to have embedded `semr` features at all the levels. This linguistic structure can be passed without any modification to the syntactic realization component.

This arrangement, however, implies a slight complication: the lexicon is indexed by the conceptual elements, but the conceptual elements are embedded in the layered representation given in input to the lexical chooser by the evaluation functions. The following configuration of features illustrates the problem:

```
Input FD: (;; toplevel is a linguistic category which is still unknown
           (semr ((cat relation) ...)))

Lexicon: ((alt (((cat relation) ...)
                ((cat set) ...) ...)))
```

In the original input, the conceptual representation is embedded under the semr feature. Nothing is known yet about its linguistic realization, so no other feature is present. The lexicon contains an alternation with an entry for each conceptual category. But the two FDs do not unify immediately: there is no conceptual cat at the toplevel of the input, so no branch can be selected.

To resolve this situation, the lexicon has been built around the following assumptions:

- Lexical entries always appear embedded under the semr feature of the linguistic constituent realizing them.
- Lexical entries set the value of linguistic features in the embedding feature, using the following notation:

  ```
  ;; Realize a set by an NP:
  ((cat set)
   ({^} ((cat np) ...)))
  ```

  That is, the link to the realization constituent is established by following the path {^} which points to the embedding feature.

- For each conceptual element, a lexical entry must provide the following information:
  - The head of the conceptual element is realized by the embedding linguistic constituent, and an open lexical item must be provided to realize it. This item is called the lexical head of the constituent.
  - The subconstituents of the conceptual element must be mapped to linguistic constituents governed by the lexical head. The linguistic structure is built incrementally, whenever a lexical head is selected. This is implemented by using a `cset` statement, as in the following example:

```
;; An assignments relation is realized
;; by a clause with head verb require:
((cat relation)
 (name assignments)
 ({^} ((cat clause)
       ;;provide lexical head
       (process ((type lexical) (lex "require")))

       ;; Build linguistic structure down one level
       (lex-roles ((requirer ((semr {^3 semr 1})))
                   (required ((semr {^3 semr 2}))))))))

 ;; Identify governed linguistic elements
 (lex-cset ((+ {^2 lex-roles requirer}
               {^2 lex-roles required})))))
```

    In this example, the relation `assignments` is mapped to a clause whose lexical head is the verb *require*. In addition to selecting the lexical head, this entry also builds the linguistic structure down one level, by constructing the `lex-roles` feature, and for each new linguistic subconstituent identified, the entry establishes the mapping to a conceptual subconstituent by filling in the semr feature. Finally, the `lex-cset` feature identifies the linguistic elements which are governed by the new lexical head, so that the recursion can proceed, and each one is lexicalized in turn.

The last complication is that, given this traversal strategy, the lexical chooser visits the linguistic constituents before their category is known. Thus, whenever a linguistic constituent is visited, there is no cat provided and the unifier cannot use the cat index to select the appropriate branch. To avoid this complication, the category labels for the conceptual elements and the linguistic elements are distinct: `sem-cat` is used for the conceptual elements, and `cat` is used for the linguistic constituents. The following entry is added at the top of the lexical chooser grammar:

```
((sem-cat linguistic)
 (lex-cset ((= semr))))
```

In this way, whenever a constituent is unified with the grammar which does not contain a sem-cat, it is unified with this ‘‘catch-all’’ branch and the recursion proceeds to the `semr` feature, which contains the conceptual information. Thus, linguistic constituents are just ignored during the lexical chooser traversal, and only the conceptual representations are acted upon.

7.6.4. Pronominalization

Usage of pronouns is required in any practical generation system. Pronominalization is a very complex issue, which I haven’t investigated. I have, however, implemented a very simple strategy for pronominalization which proves sufficient in this simple domain. The unique heuristic used to determine if an NP is to be pronominalized is to check whether the conceptual element has been realized before in the paragraph. That is, first references are full NPs, all subsequent references are pronominalized.

This simple heuristic is difficult to implement in pure FUF because ordering constraints across constituents are not explicitly represented. That is, one cannot test if a given NP occurs before or after another one which is not in the same constituent. Since I want the heuristic to work across a whole paragraph, and to be able to check if any NP occurs before any other one, this limitation prevents the implementation of the heuristic in pure FUG.

To address this issue, I have implemented the test using the `external` construct. The function `list-np-index` when applied to a total FD returns a list of pairs of the form $((path_1\ index_1)\ \ldots)$ where each $path_i$ is a

position in the total FD where an NP occurs, and `index`$_i$ is the index of this NP (recall that each role filler in the conceptual network is identified by a unique index). The list follows the linear order of occurrence in the linearized FD (`list-np-index` shares a lot of code with the FUF's linearizer). The external function `pronominalize` checks whether the index of the current NP appears before the current position. When it is the case, the FD `((pronominalize yes))` is returned.

A delicate issue is to determine when the function `list-np-index` can be called: it requires all patterns to have been instantiated, otherwise its result is not reliable. To avoid calling this function before all NPs are linearly placed in the discourse structure, FUF's `wait` construct is used. The linear order of the constituents within a clause is not to be computed until the syntactic realization component is activated. The alt determining whether to pronominalize an NP is, therefore, placed in the NP branch of SURGE, and is annotated by a `wait` construct which freezes its evaluation until the pattern of the embedding clause is instantiated.

This simplistic approach does not take into account discourse structure, quantification and syntactic structure in the decision to pronominalize. But it does generate most of the pronouns expected in fluent text and it illustrates the flexibility of the extensions to the FUG formalism described in Chap.4 (wait and external).

7.6.5. Summary

The lexical chooser of ADVISOR II implements the lexical choice techniques described in Chap.6. In addition, the implementation includes a simple form of pronominalization and the ability to realize purely argumentative clauses.

To implement the lexical chooser, I have adopted a representation of a layered structure where the conceptual elements are embedded under the linguistic constituents realizing them. Lexical entries directly map constituents elements onto linguistic information by setting features in the FD embedding the current constituent, using the `({^} x)` notation. Lexical entries fulfill two functions: they provide a lexical head for a given conceptual element, and they build the linguistic structure one level down the current constituent. Under this approach, the output of the lexical chooser is an FD describing a linguistic structure, and forms an appropriate input to the syntactic realization component without any further processing.

Finally, the most innovative aspect of the lexical chooser is the treatment of AO as a floating constraint. The AO is initially attached to the head of the discourse segment, and is percolated down the linguistic structure until an appropriate site to realize it is found. This strategy is made efficient by using FUF's control facilities.

7.7. Conclusion

The goal of the implementation of ADVISOR II was to test the lexical choice method presented in Chap.6 in a complete generation system. The lexical chooser significantly extends the range of linguistic forms that the generator can produce, leading to more fluent text. The main question raised by its design is: can a practical content generation module provide the lexical chooser with all the information it needs to make choices between alternative surface forms in a non-random manner? In other words, what is the cost of the added flexibility of the lexical chooser in terms of content determination?

The implementation of ADVISOR II demonstrated that the features required to control the lexical chooser can be derived in a natural way from a knowledge base written in an expressive formalism like CLASSIC, a user model and an evaluation system explicitly bridging the gap between objective, non-scalar information in the knowledge base, and the argumentative evaluations manipulated in language.

I have identified that some form of extra-knowledge, besides evaluation functions, is necessary to make the system function. This additional knowledge would authorize the system to rank and analyze the trade-offs between argument chains.

The main limitation of the implementation is the absence of a full text planner, capable of making the ''hard'' decisions between contradictory argument chains. In addition, the rhetorical structure of the output could be made

more varied and criteria should be found to use devices such as concession. Implementation limitations, such as the limit to two argument chains in the paragraph and the limit of three propositions in each chain should also be addressed. Most of these limitations are directly related to the absence of a full text planner.

Chapter 8
Conclusion

This research has presented new techniques to perform lexical choice that allow for more fluency in generation, a large portable reusable realization grammar and a new formalism to implement generation systems. In this chapter, I summarize the main contributions of this work and discuss how to evaluate these contributions. I then list the main limitations of the approach and of the current implementation and discuss briefly aspects of the work which could serve as a basis for future research.

8.1. Lexical Choice and Fluency

This thesis has presented techniques to improve the fluency of generated text. Three factors have been identified that contribute to increased fluency:

- Paraphrasing power: the ability to produce a variety of surface forms to express the same information.
- Compactness: the ability to combine several conceptual elements into a single linguistic constituents.
- Sensitivity: the ability to adapt generated text to a variety of speech situations.

These three goals have been addressed by specifically studying how an argumentative intent can be realized in language. The goal to argumentatively evaluate an entity can be realized at different sites in the linguistic structure. I have focused on five linguistic devices capable of realizing evaluations: judgment determiners, predicative scalar adjectives, non-predicative pre-modifiers, connotative verbs and argumentative connectives. The study of evaluative expressions therefore increases the coverage of the generator by providing a motivation to select these five types of linguistic constructions. Because the same evaluation can be realized in different ways, this study also increases the paraphrasing power of the generator. Because an evaluation can be merged with a declarative clause, the study also increases the compactness of the generated text. Finally, because evaluations correspond to the speaker's argumentative goals, the generation of evaluative expressions makes the generated text more sensitive to its interpersonal function. In summary, thus, the generation of evaluative expressions contributes significantly to a more fluent text generation.

This research has mainly focused on lexical choice as a source of linguistic variety. This position relies on the following perspective: lexical choice is the stage in the generation process where domain-specific conceptual structures, which are not motivated by linguistic considerations, are mapped onto domain-independent, linguistic structures. This structure-to-structure mapping is made complex by the fact that conceptual and linguistic structures are not isomorphic, and the same conceptual element can be mapped at different sites in a linguistic structure. I have called such conceptual elements *floating elements*. The consideration of floating conceptual elements during lexical choice allows for structural paraphrasing and significantly increases the flexibility of the generator.

The integration of argumentative constructs in generation is discussed in Chap.2. The new lexical choice techniques allowing the selection of evaluative devices are presented in Chap.6. An ontology for representing a non-linguistically motivated input to the lexical chooser is also described in the same chapter.

8.2. Reusable Generation Components

One of the goals of this research is to consolidate existing work in generation and make it more readily accessible to other researchers as well as application developers. In this spirit, I have developed FUF, a programming environment specialized to develop generation applications and SURGE a large, linguistically-founded syntactic realization component.

These generation components are portable and reusable. They have been tested and used in more than 30 research labs worldwide, and they are being actively used for more than 12 distinct research projects. The FUF formalism is presented in Chapters 3 and 4. It is an extension and implementation of the Functional Unification Grammar (FUG) formalism introduced by Kay [Kay 79]. The SURGE grammar is presented in Chap.5. It is probably the largest unification-based generation grammar developed to this day. The development of SURGE has insured the robustness of the FUF implementation. In turn, extensive experimentation with SURGE by many researchers has demonstrated the robustness of the SURGE grammar, its portability and usability.

Appendix A discusses the coverage of the current version of the SURGE grammar.

8.3. Extended Generation Formalism

The work presented in this thesis has been mostly implemented using a specialized formalism for generation. FUF is a typed unification-based formalism which is an extension of the Functional Unification Grammar formalism of [Kay 79]. The new features embodied in FUF make it a more expressive, more usable and more efficient formalism, while preserving its simple and elegant semantics. New features include typing, sophisticated control mechanisms and constructs to enhance the modularity of grammars.

The base FUG formalism is presented in Chap.3. It is first compared to existing generation formalisms: NIGEL and MUMBLE. The main advantages of FUG over these formalisms are:

- Uniform representation (FDs) and simple mechanism (unification) to manipulate both input and grammars.
- Bidirectionality and constraint propagation: allow an easy implementation of a grammar capable of sophisticated syntactic inference (cf. p.126); avoid imposing an artificial distinction between decision making and realization.
- Use of partial information: input to the generator can be partially specified; the grammar can fill up the missing part by using constraint propagation or by using defaults. Input can contain mixed specifications, including simultaneously conceptual and linguistic constraints.
- Flexible order of decision making: since order of decisions in generation depends on the input, it is important to have the ability to dynamically infer an acceptable order of decisions, regardless of the order in which the constraints are written in the grammar. FUF is capable of infering such an acceptable order of decisions through search and backtracking.

Because in ADVISOR II the same constraints, namely argumentative relations called topoi, are used both at the conceptual level and at the linguistic level, it was important to use the same formalism to uniformly encode lexical choice (map conceptual representations to a semantico-linguistic form) and syntactic realization (map a linguistic-form to a syntactic form). The implementation of lexical choice in FUF has given rise to new needs which I have addressed by extending the FUG formalism. The new extended formalism is called FUF and is presented in Chap.4. In particular, the FUF extensions address the following issues:

- **Efficiency**: lexical choice in ADVISOR II and the expression of evaluative expressions required FUF to be able to handle floating conceptual elements. Input configurations containing floating elements produced challenging search problems for the unifier. The original chronological backtracking mechanism proved too inefficient to handle such tasks. I have therefore developed four new control tools that dramatically improve FUF's efficiency. The general strategy to constrain the control strategy used by the unifier is to add annotations to the grammar that declare how to dynamically order decisions to minimize search. The control tools defined in FUF include indexing, dependency-directed backtracking (with bk-class), goal delaying (with wait) and conditional evaluation (with ignore).

- **Usability**: I have found that the expression of common linguistic constraints, such as taxonomic relations between features or the expression of completeness constraints on a constituent, are difficult or impossible to express correctly in the original FUG formalism. Furthermore, as the grammar grows, especially when lexical information is included in the grammar, its organization, readability and maintainability become important. I have addressed these problems by developing a comprehensive type mechanism for FUGs. Types address the limitations of the original formalism in an elegant manner. They also allow for a more concise organization of the grammar, taking advantage of inheritance, improve readability and ease grammar maintenance.
- **Modularity issues**: as more of the generation process is encompassed within the scope of the FUF formalism, certain constraints have become difficult to express using only the original FUG formalism. Thus, interaction with external knowledge sources such as the knowledge base or the user model becomes necessary when dealing with lexical choice. I have developed a tool that allows the interleaving of decision making in FUF and in external knowledge sources while leaving the overall control to the FUF grammar. This facility allows the development of a generator in a modular architecture where the total FD being constructed plays a role similar to a blackboard in a blackboard architecture. Another consequence of the growing size of the grammar is the difficulty of maintaining it in a consistent state. I have developed a syntax for writing grammars in a modular way, allowing the user to define abstract alternations and conjunctions and reuse them by name in different contexts.

Because the resulting formalism is rich and efficient enough, it has been possible to use it uniformly to perform more generation tasks: part of content organization and lexical choice in addition to syntactic realization.

8.4. Evaluation

This section discusses different ways of evaluating the work presented in this thesis. Evaluation of natural language systems is notoriously difficult. I present here ways of testing only certain aspects of this work and of measuring its significance. The evaluation of each main contribution is discussed separately:

- Lexical choice
- Reusable syntactic realization grammar
- Generation formalism

In each case, I list here ways to convince a skeptical reader that the presented techniques (1) are preferable to previously existing ones and (2) perform a valuable and generally useful service.

An evaluation of each point can be done along different paths. The first one is a comparative evaluation: the presented lexical chooser, grammar and formalism can do most of what existing ones can do, plus some new things. The second path is a comparison between the generated output and a target output, an existing corpus of text. The third path is a controlled evaluation of the generated output by human subjects. Finally, for the evaluation of the formalism, factors such as speed and expressiveness can be quantitatively measured and evaluated directly. For each point, a different evaluation method is most appropriate. These methods are discussed in the following subsections.

8.4.1. Evaluation of the Lexical Choice Techniques

This thesis presents a lexical chooser with three claimed advantages:

1. **Larger coverage**: (1) syntactic classes that were not addressed in previous work are now generated and as a consequence (2) the generated text covers more of the words and constructs observable in real text.
2. **More flexible**: the lexical chooser can produce from the same conceptual input (logical form) different realizations, when certain pragmatic features vary. In previous work, these different linguistic realizations required different conceptual input specifications (the logical forms had to be changed). The main advantage of a flexible surface generator is that (1) it relieves the component in charge of

building logical forms from the burden of selecting one ‘‘optimal’’ logical form among different logical forms which convey the same information but achieve different communicative effects; (2) it produces more varied and less mechanical output.

3. **More sensitive to pragmatic goals**: the lexical chooser selects different evaluative expressions depending on an input argumentative goal, thus producing more appropriate output in various situations.

The first point (larger coverage) can be first evaluated comparatively. Previous systems did not cover the generation of evaluative expressions - scalar adjectives, connotative verbs and judgment determiners. In addition, previous systems did not perform complex NP planning and choice of clause perspective (except for the recent Joyce system [Rambow & Korelsky 92] which addresses the issue of clause planning). So this is significant progress over existing lexical choosers.

This progress, however, must also be proven to be valuable in complete systems. This is best evaluated by comparing the generated output with an existing corpus of text. I have performed some of this evaluation using a corpus of 40,000 words and 3,500 sentences, which is a transcript of advising sessions recorded at Columbia. The transcripts cover approximately 6 hours of advising for 6 different advisors and 25 different students. A first quantitative analysis of the corpus measures the number of distinct words of each part-of-speech. This is shown in Table 8-0. This analysis was performed using an automatic part-of-speech tagger and some tools developed for the XTRACT corpus analysis system [Smadja 91a].

Part of speech	# Occurrences	# Distinct elt	# Elt with >2 occurrences
Nouns	5541	753	335
Verbs	3732	307	135
Adjectives	1412	243	66
Adverbs	2336	129	75
Determiners	3000	9	9
Personal pronouns	3832	19	19
Modals	981	11	9
Be	1851	1	1
Prepositions	2159	52	38
Connectives	2119	24	19

Figure 8-1: Distribution of words in corpus

In terms of coverage, the last column of the table indicates the number of ‘‘frequent’’ words in the domain, *i.e.*, the words appearing more than twice over the corpus. Although the corpus is quite small, this number appears to be significant. This was tested by performing the same analysis on each half of the corpus for the open-class categories (nouns, verbs, adjectives and adverbs). The same most frequent words were found in each half in more than 90% of the cases. This indicates that this list of some 800 most frequent words forms a good target of which words should be included in a generation system for this domain.

The following figures measure the increase in coverage allowed by the analysis of evaluative expressions: 480 of the 2119 occurences of connectives (22%) are argumentative connectives (defined here as a member of the list *but, so, because, since, although, though*). 348 out of the 1412 occurences of adjectives (24%) are scalar adjectives corresponding to the scales defined in ADVISOR II. 300 out of the 3000 occurences of non-empty determiners (10%) are judgment determiners (defined as a member of the list (*many, few, all, no, a lot, a large number of, lots of*). More convincing is the comparison between judgment determiners and cardinal determiners (*i.e., 6 assignments* vs. *many assignments*): 300 judgment determiners were used vs. only 210 cardinal determiners - that is, when a speaker had to refer to a quantity, a judgment determiner was selected more often than an exact cardinal. These figures

indicate that the generation of evaluative expressions increases the coverage of ADVISOR II by making it capable of generating 20% to 24% of all observable adjective and connective occurences in a target corpus and of generating judgment determiners which are the preferred way of referring to quantities.

These numbers measure the importance of evaluative expressions in the selected domain. A second aspect is to evaluate to which extent the current implementation takes advantage of the presented lexical choice techniques to actually increase the coverage of ADVISOR II. The current implementation of ADVISOR II covers only 200 of the 800 frequent words observed in the target corpus but it does cover 12 of the 24 observed scalar adjectives accounting for 194 of the 348 occurrences (55%) of scalar adjectives, it covers all the observed argumentative connectives and judgment determiners. The remaining words of the target corpus, which are not covered in the current implementation, are mainly nouns (280 of the list of target nouns are not described in ADVISOR II), adverbs, which are not generated at all in ADVISOR II (75 adverbs), and verbs (100 of the target verbs are not described). The main difficulty in extending the coverage of ADVISOR II would be the description of more verbs - as verbs require the most complex description in the lexicon. Overall, even in its prototype status, the current implementation covers a good proportion of the observed evaluative expressions.

A quantitative evaluation of the importance of clause planning and NP planning is more difficult to achieve. The issue is what syntactic structures can be generated *because* of clause and NP planning. A comparison with the target corpus would require a complex syntactic analysis of the corpus, which would be quite time consuming. Beyond its role in improving coverage, however, phrase planning is mainly responsible for improving the flexibility of the generator and its paraphrasing power, as discussed now.

8.4.1.1. Evaluation of the Increased Flexibility

The second claimed advantage of the presented lexical chooser is that it provides flexibility and paraphrasing power to a generator. To demonstrate how much paraphrasing power is actually achieved, I have listed in Appendix D and E the effect of clause and NP planning: the same input representation of a clause is realized in 4 different clause structures, and the same NP input representation in 30 different ways.

The presented lexical chooser can thus generate a wide variety of linguistic realizations from the same input specification. This service is valuable because (1) it creates variety in the output text, (2) increases the similarity of the generated text with the target corpus. This second point is difficult to measure quantitatively. This would require identifying ‘‘near-synonymous’’ sentences in the corpus and measuring the number of ways of expressing the same message. The small size of the available corpus makes this sort of measurement not very significant.

8.4.1.2. Evaluation of the Increased Sensitivity

The third and last claimed advantage of the lexical chooser is that it can produce ‘‘more appropriate’’ output. Appropriate is defined here very narrowly as ‘‘which clearly conveys the argumentative intent of the speaker’’. Quantitative measurement of this last point requires using user testing. Such testing has not been performed, but I present here one way it could be achieved.

The form of the test would be as follows: ADVISOR II would be used to generate a set of paragraphs describing classes with three argumentative intents set: take the class, do not take the class and no argumentative intent. For each paragraph, different user models can be randomly varied (experience of the user in different domains and types of assignments, interest in different topics). The test consists in measuring whether the input argumentative intent of the text can be reliably recovered by reading the course descriptions. An example of test paragraphs is shown in Fig.8-2. In all test paragraphs, the ‘‘last sentence’’ which provides an explicit evaluation of the course is removed (for example, *‘‘I would not recommend the course’’* is not presented to the subjects).

The expected results of such a test are that the argumentative intent can be reliably recovered, even though the paragraphs satisfy two communicative goals at once (provide the hearer with information about the class and convey an evaluation of the class) and the explicit evaluation of the course is not shown. For the evaluation of paragraphs conveying no argumentative intent, the expected result is that the evaluation of the course will remain ambiguous - that is, stating objective facts about a class should not determine its evaluation. To evaluate this point, it is unclear

If an advisor provides a student with the following description of the AI course, do you think that:
1. He thinks AI is a good course for the student.
2. He thinks AI is not a good course for the student.
3. He is not biased about the appropriateness of AI for the student.

Paragraph 1:
AI covers logic, a very theoretical topic, and it requires many assignments. So it could be difficult.

Paragraph 2:
AI covers many interesting topics, such as NLP, Vision and Expert Systems. And it involves a good amount of programming. So it should be interesting.

Paragraph 3:
AI covers logic, NLP, Vision and Expert Systems. It has 6 assignments. 5 assignments involve programming.

Figure 8-2: Example test to measure the generator's sensitivity

whether the test should allow for answers like ''don't know'' from the subjects, or force them to take position on the ''take/don't take'' issue. A detailed experiment preparation should evaluate how many subjects and paragraphs are required to make this test significant.

8.4.2. Evaluation of SURGE

Evaluating the SURGE surface realization grammar involves measuring its coverage, robustness and how much over-generation it allows. Since one of the main claims of this work is that reusable generation components can be made available, SURGE's usability must also be evaluated; that is, how easy it is to interface SURGE to different knowledge representation and lexical choice systems.

SURGE's coverage is illustrated in Appendices A and B for the clause and determiner levels. A quantitative evaluation of syntactic coverage is difficult to achieve. One possible technique would be to extract a sample of sentences from a corpus and measure which proportion of the sample can be generated by SURGE; *i.e.*, for which sentences a SURGE input can be constructed. The corpus of advising sessions discussed above does not constitute an appropriate target for such a test because it is a corpus of oral conversations, and the syntax of oral communication is quite different from that of written communication. More domain independent corpora of written text would constitute a more appropriate target. One aspect of such a test which is difficult to quantify is how ''natural'' is the encoding of the sample sentences and how valid is the syntactic analysis required for the encoding. Metrics such as number of features required, number of constituents, appropriateness of the defaults provided by SURGE would come here into play.

While such a formal sample-based evaluation of SURGE has not yet been performed, the development of SURGE has been driven in a large part by the demands of complete generation systems. At the time of writing, approximately 10,000 distinct sentences have been generated with SURGE, to be used in different applications. A core set of 2,000 sentences systematically covering most of SURGE's system has been designed and maintained for regression-testing. Such extensive testing, by a group of many different users (SURGE has been used by a group of ~ 40 researchers working on generation systems and has been used by a group of ~ 250 students in NLP classes in different universities), has proven the robustness of the grammar and served to debug it in situations which were not anticipated during its design. At this point, SURGE, while still evolving and expanding, constitutes a solid and stable surface realization grammar responding to many of the demands of complete generation systems.

The second aspect of SURGE's evaluation is how usable it is. This point has been mostly evaluated through experimentation - embedding SURGE into 5 different complete generation systems at Columbia, and making SURGE available to other researchers. SURGE has been interfaced to a wide variety of knowledge representation systems (including KL-ONE type such as LOOM and CLASSIC and simple frame systems). The functional description formalism, extended by FUF's typed features, has proven an easy target for KR systems which rely on object-based representations and inheritance. SURGE is currently being used as part of an industrial application of natural language generation at Bellcore - the PLANDOC report generation system. In this context, it has been possible to use SURGE mostly as a blackbox component in a larger architecture. The ability to ''plug and use'' SURGE is a good indication of its overall usability.

8.4.3. Evaluation of FUF

The final aspect of this work is the development of the FUF generation formalism. This formalism can be evaluated along three dimensions: expressiveness, efficiency and usability. These three aspects have been discussed in Chap.4.

The expressiveness of the base formalism has been demonstrated through the development of the SURGE grammar, of ADVISOR II's lexical chooser and through the examples of non-linguistic applications such as the list manipulation examples provided in Chap.3 and the media-coordinator of the COMET system. FUF's expressiveness has been enhanced by the addition of types and tools for the expression of completeness constraints.

Efficiency has been discussed in Sect.4.1. The effect of FUF control tools - indexing, intelligent backtracking and goal delaying - have been measured quantitatively and found significant.

Finally, useability of the formalism depends on the speed of the implementation, its tracing and debugging environment and support for modularity. FUF's usability has been evaluated empirically: a large group of users has been exposed to FUF through the use of FUF in NLP classes, and a group of 7 ''heavy users'' of FUF have been programming in FUF for several years. The suggestions of these users have driven the evolution of the system - resulting in particular in improved modularity and tracing facilities. More work remains to be done on the issue of tracing and debugging, but, as it stands, the current formalism has consistently proven to be easy to teach and to use.

8.5. Limitations

The main limitations of this work in terms of lexical choice concern the notion of degree and its use in evaluative expressions and the issue of portability of lexical information.

As far as the formalism is concerned, the development of a complete generation system has highlighted certain tasks which are not easily implemented in a declarative unification-based formalism.

Finally, the least developed aspect of this work is discourse planning: content determination, topic progression and paragraph organization. The following subsections develop each of these issues in more detail.

8.5.1. Notion of Degree

In the analysis of argumentative evaluation done in this work, I have used a tri-state system: an entity can be evaluated as either + (high on the scale), - (low on the scale) or none (no evaluation on the scale). This system is sufficient to justify a large number of linguistic distinctions and has the enormous benefit of avoiding a theoretical commitment on the notion of degree. The notion of degree, however, cannot be avoided when discussing argumentation and evaluative expressions. For example, in the current system, there is no way to distinguish between *interesting*, *very interesting* and *somehow interesting* or between *a few* and *some*.

There are many problems with defining scales with a more refined notion of degree. Horn has proposed a linguistic

test to identify if linguistic expressions are distinct degrees of the same scale [Horn 72]: if the expression *X if not Y* is acceptable, then *X* and *Y* are two degrees on the same scale. For example, *many if not most*, *most if not all* are felicitous expressions, which leads to the conclusion that there is a scale *many* < *most* < *all*. Note, however, that this scale does not correspond to the conceptual scale of quantity: * *none if not few* is not acceptable. The converse expression *few if not none* is acceptable, showing that on a different scale, the following ordering holds: *few* < *none*. For simple cases, it thus seems necessary to distinguish between conceptual scales (*e.g.*, quantity, size, complexity) and linguistic scales. It seems that expressions identified as monotonically decreasing in Sect.6.6.4 are ranked on a separate linguistic scale oriented towards the low end of the conceptual scale, while monotonically increasing expressions are ranked on the linguistic scale oriented towards the higher end of the conceptual scale. Thus for quantity, the high scale of determiners is *a few, some, many, most, all* and the low scale would be *few, none*. This distinction still does not explain why the expressions *? a few if not some* and *? some if not many* are awkward (judgments vary on these forms). It also does not explain why * *a few if not many* is not acceptable. It therefore seems that the notion of linguistic degree is distinct from the conceptual notion of ordering.

Another problem with the notion of degree is illustrated by trying to explain the ranking of items on the */many, most, all/* scale. The problem is that, while the ordering *many* < *most* < *all* seems very intuitive, the definition of semantic conditions on the usage of each term involves very different criteria:

many ≡ cardinality > typical-size
most ≡ cardinality > reference-size/2
all ≡ cardinality = ref-size

The problem is that *many* involves a comparison with a pragmatically defined typical size while *most* involves a comparison with a reference size for the quantified set.[63] So one can conceive situations where *most* is valid but *many* would not be an acceptable. For example, in a stadium typically filled with 10,000 spectators, consider a game attended by three people, two of whom feel that the game was bad. *Most spectators hated the game* is a faithful account of the situation, while *many spectators hated the game* would be misleading. In this situation, it seems that the ordering would be *most* < *many*. It is therefore unclear whether the traditional ordering *many* < *most* must be part of the definition of the lexical items *many* and *most*, or whether it is a default ordering derived because valid in most situations.

Yet another problem with the notion of degree is the definition of intensifiers like *very*, *extremely* and mitigators like *somehow* and *quite*. In fuzzy logic, vague predicates are represented as numerical distributions (using functions whose value is between 0 and 1) [Zadeh 84]. The effect of the intensifier `very` on a vague predicate characterized by the distribution p is to define a distribution p^2. A downtoner like *quite* has for effect to define a distribution $\sqrt{p}$. This account is quite counter-intuitive as it requires the definition of precise mathematical distributions to account for what feels like discrete variations. A somewhat more attractive analysis along the lines proposed in [Klein 80] is to interpret intensifiers as follows:

/few/ = less than a pragmatically defined comparison class.
/very few/ = less than /few/
/extremely few/ = less than /very few/

In this approach, intensifiers act on a pragmatically defined comparison class. If *few* compares the cardinality of a set with a certain reference set, then *very few* compares it with a cardinality that would qualify as *few*.

This semantic analysis still does not provide guidelines for a generator to decide to use an intensifier. An approach which could lead to an implementable procedure is proposed in [Jayez 88]. The idea is to define degree in epistemic terms, as the number or strength of inferences that can be derived from a predication. For example, if the number of assignments in a class supports three conclusions (*e.g.*, the class is difficult, time consuming and boring) then this number acquires a higher degree than in a situation where only one conclusion can be derived from the same evaluation.

While the notion of degree remains difficult to formalize, the absence of an analysis of degree prevents the system from reasoning about pragmatic effects like scalar implicatures [Hirschberg 85], understatements and hedges [Huebler 83].

---

[63] The interpretation of most as more than half is the classical one, given for example in [Barwise and Cooper 81] and [Keenan and Stavi 86].

8.5.2. Acquisition and Validation of Argumentative Knowledge

The ADVISOR II system makes use of a set of scales and a topoi-base to model the argumentative aspect of its task. The issue of how this argumentative knowledge can be acquired and validated in general remains to be studied. Section 2.3.2 discusses the technique of the semantic differential, an empirical psychological test to discover and validate scalar dimensions in a domain [Osgood, Suci and Tannenbaum 57]. I have also discussed how a corpus analysis of scalar adjectives and argumentative connectives can provide data to support the construction of the required argumentative knowledge. Experimentation with such techniques needs to be pursued.

8.5.3. Portability of Lexicon

In this work, the lexical chooser plays the role in the generator's architecture of interfacing a conceptual knowledge representation with the linguistic structure. This position has the benefit of clearly identifying the issue of structural paraphrasing and its lexical aspect. One consequence of this position, however, is that the lexicon becomes extremely domain-dependent. A new lexicon must be built for each domain ontology.

It is clear that lexical choice is much more domain dependent than syntactic realization: domain specific idioms are frequent, the semantic interpretation of lexical items is also domain dependent. There are, however, aspects of the lexicon that are domain independent and could be isolated in a reusable component. For example, the lexical properties of adjectives discussed in Sect.6.7 can be encoded in a portable lexicon. The mapping from adjective to scale would remain domain-specific.

The study of which aspects of the lexicon can be made portable and what effect this separation would have on the lexical chooser remains to be done.

8.5.4. Pronominalization, Reference Planning and Clause Combining

The FUF formalism has been found adequate to implement the structure-to-structure mapping aspect of the lexical chooser. The control flow enforced during this mapping consists in a mainly top-down traversal of the linguistic structure being built. There are, however, aspects of the generation task which cannot be modelled as a hierarchical structure traversal. Among these are pronominalization, reference planning and clause combining. This point was also raised in [Reiter and Mellish 92].[64]

These tasks are difficult to implement using a formalism like FUF because they require a sort of scanning of the linear structure of the text. For example, to plan referring expressions it is important to distinguish between first and subsequent reference and to identify opportunities to add attributive information on each element of a sequence of referring expressions in a paragraph. Clause combining is the process of changing sequences like *John ate an apple, Mary ate an apple* into *John and Mary ate an apple*. SURGE is capable of performing ellipsis, as explained in Sect.4.1.5.1. The implementation can recognize a case of ellipsis when two clauses containing common elements appear successively in a conjunction. This approach is perfectly sufficient under the assumption that a paragraph planner orders propositions before sending them to the lexical chooser. If this task of the paragraph planner is to be integrated into a FUF-based system, a more general approach would be required: the system would need the ability to scan an unordered set of clauses, recognize shared arguments and order the arguments accordingly to optimize the conciseness of the paragraph.

This linear scanning of the linguistic structure is difficult to implement using FUF because (1) it can involve an arbitrary number of elements (several NPs, several clauses), (2) FUF is not efficient at manipulating lists and sets and (3) linear ordering is not easily accessible in FUF (the pattern mechanism is mostly external to the core FUF formalism). Note that this task is difficult to express in most existing declarative formalisms. It seems more natural to express it in a procedural and iterative manner. However, designing an appropriate declarative formalism for

[64] I want to thank Ehud Reiter for making me understand this limitation through long discussions.

scanning and handling such non-structural relations is an interesting future direction of research. In this perspective, Robin's work on revision [Robin 92b] offers a promising approach to this issue: in his system, reference planning can be performed by first drafting an initial text, complete with regular referring expressions and then revising this draft to take advantage of the overall text structure and distribute information across different expressions referring to the same entity. Since, in this case, the whole paragraph is available when performing revision, tools derived from FUF can be used to scan the whole structure.

8.5.5. Paragraph Planning

The ADVISOR II system contains a very limited form of paragraph planning. In particular, it does not implement the trade-off and decision analysis required to choose among contradictory arguments at the content determination stage, it does not use explicit content organization criteria to determine in which order arguments should be presented, and finally, it does not implement rules of topic progression. Just following the argument chains does produce coherent paragraphs, but more attention to this aspect would significantly increase variety in the output of the system.

Another limitation of the paragraph planner is that the evaluation functions are limited to intrinsic evaluation of classes - that is, they only consider the attributes of a single class to evaluate it. *Extrinsic* evaluation criteria would considerably enrich the content generation aspect of ADVISOR II. Example of extrinsic evaluations are: comparison of one class with another, more or less appropriate; consideration of ''aggregate'' attributes over the whole schedule of the student (such as total workload of the student, number of courses in each area of the curriculum). Such evaluation criteria would greatly enhance what it is possible to say about a course.

8.6. Future Work

This research has identified several areas for possible research. First, addressing the limitations listed above involves:

- Coming up with a workable definition of degree, that is related both to conceptual ordering and to epistemic factors, that is compositional, and allows for the generation of pragmatic effects like scalar implicature and hedges.
- Designing empirical techniques to develop an argumentative model of a specific domain.
- Distinguishing the domain-independent features in lexical entries and designing a portable reusable lexicon compatible with the lexical choice techniques presented in this work.
- Developing a declarative formalism capable of expressing relations between sequences of constituents not structurally related.
- Studying paragraph planning techniques appropriate for argumentative text.

Extensions based on this work include research on more efficient formalisms for text generation, specific content determination issues and the study of more structural lexical transformations. I develop each of these points in the following paragraphs.

8.6.1. Formalism for Text Generation

As mentioned above, this work has identified a class of tasks which are difficult to implement using a unification-based formalism of the FUF type. It has also shown that FUF can be used for more tasks than was previously considered (cf. Sect.3.4 for a discussion of FUF as a general purpose programming language).

Experience with the current implementation has also identified an efficiency bottleneck related to the data-structures used for representing feature structures and paths. A different algorithm for unification and different data-structures for feature structures need to be developed to overcome this limitation. [Ait-Kaci 84] has proposed an algorithm for

unification on Ψ-terms (a data-structure very close to functional descriptions) based on the union-find algorithm which behaves more efficiently. Similar ideas have been presented in [Huet 76] and [Escalada-Imaz & Ghallab 88]. Unfortunately, these algorithms loose their advantages when backtracking is introduced, as the union operations of the union-find algorithm need to be undone frequently. [Mannila and Ukkonen 86a; Mannila and Ukkonen 86b; Mannila and Ukkonen 86c; Mannila and Ukkonen 88] have addressed this problem in the context of Prolog unification with backtracking. It is not clear whether their techniques can be readily applied to FUF's type of unification.

8.6.2. Content Determination for Missing Features

The general methodology followed in this work to design a generator is to first analyze the linguistic variety, derive from this analysis a set of features that control the variety, and then derive the value of these features from the conceptual input available to the generator. In other words, the input to the generator is induced from an observation of how the output can vary.

With the current level of generation technology, the realization component can produce much more variation than can be accounted for by the content determination module - that is, there is still a large potential of linguistic expertise which cannot be taken advantage of by current content determination technology. This situation has been encountered in this work, for example in the case of determiners. SURGE requires 23 features to control the realization of the determiner sequence of NPs. In this work, the lexical chooser can only produce a value for 13 of these features. The other features are either produced randomly or fixed to a constant value.

To address this imbalance, one needs to (1) produce more powerful content determination techniques, which are driven by the needs of the realization component, and (2) refine the linguistic analysis to determine if the input features expected by the realization component are defined at the appropriate level of abstraction and can effectively be determined by a reasonable content determination program.

8.6.3. Nominalizations

This work has emphasized the issue of mapping a conceptual structure onto a non-isomorphic linguistic structure. Some of the decisions that must be made by the lexical chooser to accomodate the mismatch between these structures are, at the clause level, the possibility to merge two conceptual relations into a single composite process and the possibility to merge an argumentative evaluation with a clause realizing a conceptual relation; at the NP level, modifying relations can be mapped to adjectival describers, noun-noun modifiers, relative clauses or PP post-modifiers.

Other linguistic resources which have not been considered in this work can significantly increase the paraphrasing power of the generator. These include nominalization and adjuncts. Nominalization is the process of mapping a conceptual relation with its arguments onto a complex noun phrase. At the linguistic level, nominalization consists in turning a clause into an NP. Extensive linguistic analyses of the phenomenon have been proposed. Especially useful analyses from a generation perspective are presented in [Levi 78, Chap.5] and [Vendler 68]. The ability to generate nominalizations would allow a generator to select between *the Boston Celtics destroyed the Denver Nuggets 155-42, extending their winning streak to four games* and *the smashing victory of the Boston Celtics over the Denver Nuggets extended their winning streak to four games*. Such abilities have been discussed in the GOSSIP system [Iordanskaja et al 91]. A promising method to address this issue, relying on revision, is also proposed in [Robin 92b]. More research on the use of nominalizations and the control of the decision whether to nominalize would significantly increase the fluency of generated text.

8.7. Conclusion

The present work has contributed to the maturity of the field of generation by building portable reusable components for surface realization, helping other researchers to focus on new challenging tasks in the field of content determination and organization. It has provided elements to increase the fluency of generated text by improving the flexibility of lexical choice. It has also demonstrated the adequacy of a typed unification formalism to implement many generation tasks in a uniform manner.

Writing this dissertation has served as a reminder of the vastness and complexity of natural language and of its production. The many unresolved issues raised by this work should be the focus of continued research to help future generation(s).

Bibliography

[Abraham 79] Abraham, Werner.
BUT.
Studia Linguistica XXXIII(II):89-119, 1979.

[Ait-Kaci 84] Ait-Kaci, H.
A Lattice-theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures.
PhD thesis, University of Pennsylvania, 1984.
UMI #8505030.

[Allen 83] Allen, J.F.
Maintaining Knowledge about Temporal Intervals.
Communications of the ACM 26(11):832-843, 1983.

[Anscombre & Ducrot 83]
Anscombre, J.C. and O., Ducrot.
Philosophie et langage: *L'argumentation dans la langue.*
Pierre Mardaga, Bruxelles, 1983.

[Appelt 85] Appelt, D.E.
Planning English Sentences.
Cambridge University Press, Cambridge, England, 1985.

[Bartsch 89] Bartsch, R.
Semantics and Contextual Expression.
Foris Publications, Dordrecht, Holland; Providence, RI, 1989.

[Barwise and Cooper 81]
Barwise, J. and R. Cooper.
Generalized quantifiers in English.
Linguistics and Philosophy 4:159-219, 1981.

[Bateman et al. 90]
Bateman, J.A. and Kasper, R.T. and Moore, J.D. and Whitney, R.A.
A general organization of knowledge for natural language processing: the PENMAN Upper-Model.
Technical Report, ISI, Marina del Rey, CA, 1990.

[Bolinger 72] Bolinger, D.
Degree Words.
Mouton, The Hague, 1972.

[Boyer 88] Boyer, M.
Towards Functional Logic Grammars.
In Dahl, V. and Saint-Dizier P. (editor), *Natural Language Programming and Logic Programming, II*, pages 45-62. North Holland, Amsterdam, 1988.

[Brachman et al. 90]
Brachman, R.J. and Resnick, L.A. and Borgida, A. and McGuiness, D.L. and Patel-Schneider, R.F.
Living with CLASSIC: When and how to use a KL-ONE-like language.
In Sowa, J. (editor), *Principles of semantic networks*. Morgan Kaufman Publishers, 1990.

[Brew 91] Brew, C.
Systemic Classification and its Efficiency.
Computational Linguistics 17(4):375-408, 1991.

[Brown & Levinson 87]
Brown, P. & Levinson, S.C.
Studies in Interactional Sociolinguistics 4: *Politeness : Some universals in language usage.*
Cambridge University Press, Cambridge, 1987.

[Bruxelles & Raccah 91]
Bruxelles, S. and Raccah P.Y.
Argumentation et Semantique: le parti-pris du lexique.
Actes du Colloque 'Enonciation et parti-pris'.
Forthcoming, 1991.

[Bruxelles *et al* 89]
Bruxelles, S., Carcagno, D. and Fournier, C.
Vers une construction automatique des topoi a partir du lexique.
CC AI - Journal for the integrated study of Artificial Intelligence cognitive science and applied epistemology 6(4):309-328, 1989.

[Buchberger & Horacek 88]
Buchberger, E. and H. Horacek.
VIE-GEN: a generator for German texts.
In McDonald, D. and L. Bloc (editor), *Natural Language Generation Systems*, pages 166-204. Springer-Verlag, 1988.

[Chomsky 57] Chomsky, N.
Syntactic Structures.
Mouton, The Hague, 1957.

[Cohen 84] Cohen, R.
A computational theory of the function of clue words in argument understanding.
In *Coling84*, pages 251-258. COLING, Stanford, California, July, 1984.

[Cumming 86] Cumming, S.
The lexicon in text generation.
In *Proceedings of the 1st Workshop on Automating the Lexicon*. 1986.

[Dale 88] Dale, R.
Generating referring expressions in a domain of objects and processes.
PhD thesis, University of Edinburgh, 1988.

[Danlos 87a] Danlos, L.
The Linguistic Basis of Text Generation.
Cambridge University Press, Cambridge, England, 1987.

[Danlos 87b] Danlos, L.
A French and English Syntactic Component for Generation.
In Gerard Kempen (editor), *Natural Language Generation*, pages 191-218. Martinus Nijhoff Publishers, 1987.

[Danlos and Namer 88]
Danlos, L. and F. Namer.
Morphology and Cross Dependencies in the Synthesis of Personal Pronouns in Romance Languages.
In *Proceedings of COLING-88*. Budapest, 1988.

[Davey 79] Davey, A.
Discourse Production.
Edinburgh University Press, Edinburgh, 1979.

[de Kleer et al 79] de Kleer, J. and Doyle, J. and Steele, G.L. and Sussman, G.J.
Explicit Control of Reasoning.
In Winston P.J. and R.H. Brown (editor), *Artificial Intelligence: an MIT Perspective*, pages 93-116. MIT Press, 1979.

[De Smedt 90] De Smedt, K.
Incremental Sentence Generation.
PhD thesis, Nijmegen Institute for Cognition Research and Information Technology, 1990.
NICI Technical Report 90-01.

[Dispaux 84] Dispaux, G.
La Logique et le Quotidien: Une Analyse Dialogique des Mechanismes d'Argumentation.
Les Editions de Minuit, Paris, 1984.

[Downing 77] Downing, P.A.
On the creation and use of English compound nouns.
Language 53:810-842, 1977.

[Ducrot 72] Ducrot, O.
Collection Savoir: *Dire et ne pas dire - Principes de semantique linguistique.*
Herman, Paris, 1972.

[Ducrot 83] Ducrot, O.
Le sens commun: *Le dire et le dit.*
Les editions de Minuit, Paris, 1983.

[Ducrot *et al* 80] Ducrot, O. *et al*.
Le sens commun: *Les mots du discours.*
Les editions de Minuit, Paris, 1980.

[Dymetman & Isabelle 88]
Dymetman, M. and P. Isabelle.
Reversible logic grammars for machine translation.
In *Proceedings of the Second International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Language*. Center for Machine Translation, CMU, 1988.

[Elhadad 91a] Elhadad, M.
FUF User Manual - Version 5.0.
(Tech. Rep. CUCS-038-91) CUCS-038-91, Columbia University - Computer Science, 1991.

[Elhadad 91b] Elhadad, M.
Generating Adjectives to Express the Speaker's Argumentative Intent.
In *Proceedings of 9th National Conference on Artificial Intelligence (AAAI 91)*, pages 98-104. Anaheim, 1991.

[Elhadad 92] Elhadad, M.
Generating Argumentative Paragraphs.
In *Proceedings of COLING-92*. Nantes, France, July, 1992.

[Elhadad & McKeown 90]
Elhadad, M. and K.R. McKeown.
Generating Connectives.
In *Proceedings of COLING'90 (Volume 3)*, pages 97-101. Helsinki, Finland, 1990.

[Elhadad & Robin 92]
Elhadad, M. & Robin, J.
Controlling Content Realization with Functional Unification Grammars.
In R. Dale, E. Hovy, D. Roesner and O. Stock (editor), *Aspects of Automated Natural Language Generation*, pages 89-104. Springer Verlag, 1992.

[Elhadad et al. 91] Elhadad, M., Feiner, S., McKeown, K., and Seligmann, D.
Generating customized text and graphics in the COMET explanation testbed.
In *Proc. 1991 Winter Simulation Conference*, pages 1058–1065. Phoenix, AZ, December 8–11, 1991.

[Emele & Zajac 90]
M.C. Emele and R. Zajac.
Typed Unification Grammars.
In Hans Karlgren (editor), *Proceedings of COLING'90*, pages Vol 3. 293-298. Helsinki, Finland, 1990.

[Escalada-Imaz & Ghallab 88]
Escalada-Imaz, G. and M. Ghallab.
A Practically Efficient and Almost Linear Unification Algorithm .
Artificial Intelligence 36:249-263, 1988.

[Fawcett 87] Fawcett, R.P.
The semantics of clause and verb for relational processes in English.
In Halliday, M.A.K. & Fawcett, R.P. (editor), *New developments in systemic linguistics*. Frances Pinter, London and New York, 1987.

[Feiner and McKeown 91]
Feiner, S. and McKeown, K.R.
Automating the Generation of Coordinated Multimedia Explanations.
IEEE Computer 24(10):33-41, October, 1991.

[Fillmore 71] Fillmore, C.J.
Verbs of judging: an exercise in semantic description.
In Fillmore C.J. and D.T. Langendoen (editor), *Studies in Linguistic Semantics*, pages 272-289. Holt, Rinehart and Winston, New York, 1971.

[Fillmore, Kay & O'Connor 88]
Fillmore, C.J, Kay, P. and O'Connor C.
Regularity and idiomaticity in grammatical constructions: the case of *let alone*.
Language 64(3), 1988.

[Gazdar et al 85] Gerald Gazdar, Ewan Klein, Geoffrey Pullum & Ivan Sag.
Generalized Phrase Structure Grammars.
Harvard University Press, Cambridge, MA, 1985.

[Givon 70] Givon, T.
Notes on the semantic structure of English adjectives.
Language 46(4):816-837, 1970.

[Goldman 75] Goldman, N.M.
Conceptual generation.
In Schank, R.C. (editor), *Conceptual Information Processing*. North Holland, Amsterdam, 1975.

[Gross 75] Gross, M.
Methodes en syntaxe.
Hermann, Paris, 1975.

[Grosz & Sidner 86]
Grosz, B. and Sidner, C.
Attentions, intentions, and the structure of discourse.
Computational Linguistics 12(3):175-204, 1986.

[Halliday 76a] Halliday, M.A.K.
the form of a functional grammar.
In Kress, G.R. (editor), *Halliday : System and Function in Language*, pages pages 7-25. Oxford University Press, London, 1976.

[Halliday 76b] Halliday, M.A.K.
System and Function in Language.
Oxford University Press, London, 1976.

[Halliday 85] Halliday, M.A.K.
An Introduction to Functional Grammar.
Edward Arnold, London, 1985.

[Hirschberg 85] Hirschberg, J.B.
A Theory of Scalar Implicature.
PhD thesis, University of Pennsylvania, 1985.

[Hirschberg & Litman 87]
Hirschberg, J. and Litman, D.
Now let's talk about Now: identifying clue phrases intonationally.
In *Proceedings of the 25th Conference of the ACL*, pages 163-171. Association for Computational Linguistics, 1987.

[Hirsh 88] Hirsh, Susan.
P-PATR: A Compiler for Unification-based Grammars.
In Dahl, V. and Saint-Dizier, P. (editor), *Natural Language Understanding and Logic Programming, II*, pages 63-77. North Holland, Amsterdam, 1988.

[Horacek 92] Horacek, Helmut.
An Integrated View of Text Planning.
In R. Dale, E. Hovy, D. Roesner and O. Stock (editor), *Proceedings of the Sixth International Workshop on Natural Language Generation, Trento, Italy*. Springer Verlag, 1992.
To appear.

[Horn 72] Horn, L.R.
On the Semantic Properties of Logical Operators in English.
PhD thesis, University of California at Los Angeles, 1972.

[Houghton 86] Houghton, G.
The Production of Language in Dialogue: a Computational Model.
PhD thesis, University of Sussex, 1986.

[Hovy 88a] Hovy, E.H.
Generating natural language under pragmatic constraints.
L. Erlbaum Associates, Hillsdale, N.J., 1988.
Based on the author's thesis (doctoral--Yale University, 1987).

[Hovy 88b] Hovy, E.H.
Planning Coherent Multisentential Text.
In *Proceedings of the 26th ACL Conference*, pages P.163-169. ACL, Buffalo, June, 1988.

[Hovy 90] Hovy, E.
Unresolved issues in paragraph planning.
In Dale, R., Mellish, C.S. and Zock, M. (editor), *Current Research in Natural Language Generation*, pages 17-45. Academic Press, 1990.

[Hovy 91] Hovy, E.
Approaches to the planning of coherent text.
In Paris, C.L., Swartout, W.R. and Mann, W.C. (editor), *Natural Language Generation in Artificial Intelligence and Computational Linguistic*, pages 83-102. Kluwer Academic Publishers, 1991.

[Huebler 83] Huebler, A.
Pragmatics and Beyond. Volume IV:6: *Understatements and Hedges in English.*
John Benjamins Publishing Company, Amsterdam, 1983.

[Huet 76] Huet, G.
Resolution d'Equations dans des langages d'ordre 1,2,...,ω.
PhD thesis, Universite de Paris VII, France, 1976.

[Iordanskaja et al 91]
Iordanskaja L., Kittredge R. and Polguere A.
Lexical Selection and Paraphrase in a Meaning-text Generation Model.
In Cecile L. Paris, William R. Swartout and William C. Mann (editors), *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, pages 293-312. Kluwer Academic Publishers, 1991.

[Jacobs 85] Jacobs, P. S.
A knowledge-based approach to language production.
PhD thesis, Univ. of California, Berkeley, 1985.

[Jacobs 87] Jacobs, P.S.
Knowledge-intensive natural language generation.
Artificial Intelligence 33:325-378, 1987.

[Jayez 88] Jayez, J.
L'inference en langue naturelle.
Hermes, Paris, 1988.

[Johnson 88] Johnson.
lecture notes. Volume 16: *Attribute-value Logic and the Theory of Grammar*.
CSLI, 1988.

[Joshi 87] Joshi, A.K.
The relevance of tree adjoining grammar to generation.
In Gerard Kempen (editor), *Natural Language Generation: New Results in Artificial Intellligence, Psychology and Linguistics*, pages 233-252. Martinus Ninjhoff Publishers, 1987.

[Kaplan & Bresnan 82]
Kaplan, R.M. and J. Bresnan.
Lexical-functional grammar: A formal system for grammatical representation.
In Bresnan, J. (editor), *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, MA, 1982.

[Karttunen 84] Karttunen, L.
Features and Values.
In *Coling84*, pages 28-33. COLING, Stanford, California, July, 1984.

[Karttunen 85] Karttunen, L.
Structure Sharing with Binary Trees.
In *Proceedings of the 23rd annual meeting of the ACL*, pages 133-137. ACL, 1985.

[Karttunen 86] Karttunen, L.
Radical Lexicalism.
Technical Report CSLI-86-66, CSLI, 1986.

[Karttunen & Peters 79]
Karttunen, L. & S. Peters.
Conventional Implicature.
In Oh & Dinneen (editor), *Syntax and Semantics*. Volume 11: *Presupposition*, pages 1-56. Academic Press, New York, 1979.

[Kasper 87] Kasper, R.
Feature Structures: A Logical Theory with Applications to Language Analysis.
PhD thesis, University of Michigan, 1987.

[Kasper 88] Kasper, R.
Systemic Grammar and Functional Unification Grammar.
In Benson J.D and W.S. Greaves (editor), *Advances in Discourse Processes*. Number XXVI: *Systemic Functional Approaches to Discourse: Selected Papers from the 12th International Systemic Workshop*, pages 176-199. Ablex, Norwood, NJ, 1988.

[Kasper 89] Kasper, R.
A flexible interface for linking applications to Penman's sentence generator.
In *Proceedings of 1989 DARPA Speech and Natural Language Workshop*, pages 153-158. Philadelphia, 1989.

[Kasper&Rounds 86] Kasper, R. and W. Rounds.
A Logical Semantics for Feature Structures.
In *Proceedings of the 24th meeting of the ACL*. ACL, Columbia University, New York, NY, June, 1986.

[Kay 79] Kay, M.
Functional Grammar.
In *Proceedings of the 5th meeting of the Berkeley Linguistics Society*. Berkeley Linguistics Society, 1979.

[Kay 84] Kay, M.
Functional Unification Grammars: a Formalism for Machine Translation.
In *Proceedings of Coling 84*, pages 75-78. Stanford, CA, 1984.

[Kay 85] Kay, M.
Parsing in Unification grammar.
In Dowty, Karttunen & Zwicky (editor), *Natural Language Parsing*, pages 152-178. Cambridge University Press, Cambridge, England, 1985.

[Keenan and Stavi 86] Keenan, E. and Y. Stavi.
A semantic characterization of natural language determiners.
Linguistics and Philosophy 9:253-326, 1986.

[Kerbrat-orecchioni 77] Kerbrat-orecchioni, C.
La connotation.
Presses Universitaires de Lyon, Lyon, 1977.

[Klein 80] Klein, E.
A semantics for positive and comparative adjectives.
Linguistics and Philosophy 4(1):1-45, 1980.

[Knight 89] Knight, K.
Unification: a Multidisciplinary Survey.
Computing Surveys 21(1):93-124, March, 1989.

[Kukich 83a] Kukich, K.
Design of a knowldege based text generator.
In *Proceedings of the 21st ACL Conference*. ACL, 1983.

[Kukich 83b] Kukich, K.
Knowledge-based report generation: a knowledge engineering approach to natural language report generation.
PhD thesis, University of Pittsburgh, 1983.

[Lakoff 71] Lakoff, R.
IFs, ANDs and BUTs: about conjunction.
In Fillmore & Langendoen (editor), *Studies in Linguistic Semantics*, pages 114-149. Holt, Rinehart & Winston, New York, 1971.

[Lakoff 75] Lakoff, G.
Hedges: A Study in Meaning Criteria and the Logic of Fuzzy Concepts.
In D. Dockneey, W. Harper and B. Freeds (editor), *Contemporary Research in Philosophical Logic and Linguistic Semantics*, pages 221-271. D. Reidel, Dordrecht, 1975.

[Lang 84] Lang, Ewald.
SLCS. Volume 9: *The Semantics of Coordination.*
John Benjamins B.V., Amsterdam, 1984.
Original edition: *Semantic der koordinativen Verknupfung*, Berlin, 1977.

[Lascarides & Oberlander 92]
Lascarides, A. and J. Oberlander.
Abducing Temporal Discourse.
In Dale, R., Hovy, E., Rosner, D. and Stock, O. (editor), *Aspects of Automated Natural Language Generation*, pages 167-182. Springer-Verlag, 1992.

[Levi 78] Levi, J.N.
The Syntax and Semantics of Complex Nominals.
Academic Press, New York, 1978.

[Levin and Rappaport 86]
Levin, B. and M. Rappaport.
The Formation of Adjectival Passives.
Linguistic Inquiry 17(4):623-661, 1986.

[Levinson 83] Levinson, S.C.
Pragmatics.
Cambridge University Press, Cambridge, England, 1983.

[Ludlow 89] Ludlow, P.
Implicit comparison classes.
Linguistics and Philosophy 12:519-533, 1989.

[Mann 83] Mann, W.C.
An overview of the Nigel Text Generation Grammar.
Technical Report ISI/RR-83-113, USC/ISI, April, 1983.

[Mann 85] Mann, W.
The anatomy of a Systemic Choice.
Discourse Processes 8:53-74, 1985.

[Mann & Matthiessen 83a]
Mann, W.C. and Matthiessen, C.
Nigel: a Systemic Grammar for Text Generation.
Technical Report ISI/RR-83-105, USC/ISI, 1983.

[Mann & Matthiessen 83b]
Mann, W.C. and Matthiessen, C.
Nigel: a Systemic Grammar for Text Generation.
Technical Report ISI/RR-83-105, USC/ISI, 1983.

[Mann & Thompson 87]
Mann, W.C. and S.A. Thompson.
Rhetorical Structure Theory: Description and Construction of Text Structures.
In Gerard Kempen (editor), *Natural Language Generation*, pages 85-96. Martinus Nijhoff, 1987.

[Mannila and Ukkonen 86a]
Mannila, H. and E. Ukkonen.
The set union problem with backtracking.
In *Proc. 13th International Colloquium on Automata, Languages and Programming (ICALP 86)*, pages 236-243. Springer Verlag, 1986.
(Lecture Notes in Computer Science 226).

[Mannila and Ukkonen 86b]
Mannila, H. and E. Ukkonen.
On the complexity of unification sequences.
In *Proc. 3rd International Conference on Logic Programming*, pages 122-133. Springer Verlag, 1986.
(Lecture Notes in Computer Science 225).

[Mannila and Ukkonen 86c]
Mannila, H. and E. Ukkonen.
Timesstamped term representation for implementing Prolog.
In *Proc. 3rd International Conference on Logic Programming*, pages 159-167. Springer Verlag, 1986.
(Lecture Notes in Computer Science 225).

[Mannila and Ukkonen 88]
Mannila, H. and E. Ukkonen.
Time parameter and arbitrary deunions in the set union problem.
In *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT 88)*, pages 34-42. Springer Verlag, 1988.
(Lecture Notes in Computer Science 318).

[Matthiessen 85] Matthiessen, C.M.I.M.
The Systemic Framework in Text Generation : Nigel.
In Benson, J. and Greaves, W. (editor), *Systemic Perspectives on Discourse*. Ablex, Norwood, New Jersey, 1985.

[Matthiessen 88] Matthiessen, C.M.I.M.
What's in NIGEL: Lexicogrammatical Cartography.
Technical Report, USC/ISI, 4676 Admiralty Way, Marina Del Rey, CA 90292, 1988.

[Matthiessen 91] Matthiessen, C.M.
Lexicogrammatical choice in text generation.
In Paris, C. and Swartout, W. and Mann, W.C. (editor), *Natutal Language Generation in Artificial Intelligence and Computational Linguistics*. Kluwer Academic Publishers, 1991.

[McDonald 80] McDonald, D.
Natural Language Generation as a Process of Decision-Making under Constraints.
PhD thesis, MIT, 1980.

[McDonald et al. 87]
McDonald, D.,Vaughan, M. & Pustejovski, J.
Factors contributing to efficiency in natural language generation.
In Gerard Kempen (editor), *Natural Language Generation: New Results in Artificial Intellligence, Psychology and Linguistics*, pages 159-182. Martinus Ninjhoff Publishers, 1987.

[McKeown 82] McKeown, K.R.
Generating Natural Language Text in Response to Questions about Database Structure.
PhD thesis, University of Pennsylvania, 1982.

[McKeown 85] McKeown, K.R.
Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text.
Cambridge University Press, Cambridge, England, 1985.

[McKeown 88] McKeown, K.R.
Generating Goal Oriented Explanations.
International Journal of Expert Systems 1(4):377-395, 1988.

[McKeown & Elhadad 91]
McKeown, K. and M. Elhadad.
A Contrastive Evaluation of Functional Unification Grammar for Surface Language Generators: A Case Study in Choice of Connectives.
In Cecile L. Paris, William R. Swartout and William C. Mann (editors), *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, pages 351-396. Kluwer Academic Publishers, 1991.

[McKeown *et al* 85]
McKeown, K.R., Wish, M., and Matthews, K.
Tailoring Explanations for the User.
In *Proceedings of the IJCAI*. IJCAI, 1985.

[McKeown et al 90]
McKeown, K., Elhadad, M., Fukumoto, Y., Lim, J., Lombardi, C., Robin, J. and Smadja, F.
Language Generation in COMET.
In Mellish, C. and Dale, R. and Zock, M. (editor), *Current Research in Language Generation*, pages 103-140. Academic Press, London, UK, 1990.

[Mel'cuk and Pertsov 87]
Mel'cuk, I.A. and N.V. Pertsov.
Surface Syntax of English - a Formal Model within the Meaning-Text Framework.
John Benjamins, Amsterdam/Philadelphia, 1987.

[Mel'cuk and Polguere 87]
Mel'cuk I. and A. Polguere.
A formal lexicon in Meaning-Text theory (or How to do Lexica with Words).
Computational Linguistics 13(3-4):261-275, July-December, 1987.

[Meteer 89] Meteer, M. W.
The SPOKESMAN Natural Language Generation System.
Technical Report Report No. 7090, BBN Systems and Technologies Corporation, 10 Moulton Street, Cambridge, MA 02138, 1989.

[Meteer 90] Meteer, M.W.
The generation gap: the problem of expressibility in text planning.
PhD thesis, University of Massachussets at Ahmerst, 1990.
Also available as BBN technical report No. 7347.

[Meteer et al. 87] Meteer, M.W. and McDonald, D.D. and Anderson, S.D. and Forster, D. and Gay, L.S. and Huettner, A.K. and Sibun, P.
Mumble-86: Design and implmentation.
Technical Report COINS 87-87, University of Massachussets at Amherst, Ahmerst, Ma., 1987.

[Moens & Steedman 88]
Moens, M. and M. Steedman.
Temporal Ontology and Temporal Reference.
Computational Linguistics 14(2):15-28, 1988.

[Moeschler 86] Moeschler, J.
Connecteurs pragmatiques, lois de discours et strategies interpretatives: parce que et la justification enonciative.
Cahiers de Linguistique Francaise (7):pages 149-168, 1986.

[Moore & Paris 89]
Moore, J.D. and C.L. Paris.
Planning Text for Advisory Dialogues.
In *Proceeding 27th ACL*, pages 203-211. ACL, Vancouver, BC, June, 1989.

[Moore & Paris 91]
Moore, J.D. & C.L. Paris.
Discourse Structure for Explanatory Dialogues.
In *AAAI Spring Symposium on Discourse Structure*. AAAI, 1991.

[Naish 85] Naish, Lee.
Lectures Notes in Computer Science. Volume 238: *Negation and Control in Prolog.*
Springer Verlag, 1985.

[Ney 83] Ney, J.W.
Optionality and Choice in the Selection of Order of Adjectives in English.
General Linguistics 23(2):94-128, 1983.

[Nirenburg and Nirenburg 88]
Nirenburg, S. and Nirenbrug, I.
A framework for lexical selection natural lanaguage generation.
In *Proceedings of the 11th International Conference on Computational Linguistics*. COLING, 1988.

[Nogier 90] Nogier, J.F.
Un systeme de production de langage fonde sur le modele des graphes conceptuels.
PhD thesis, Univeriste Paris VII, 1990.

[Norvig 91] Norvig, P.
Techniques for Automatic Memoization with Applications to Context-Free Parsing.
Computational Linguistics 17(1):91-98, 1991.

[Osgood, Suci and Tannenbaum 57]
Osgood, C.E., Suci, G.S. and Tannenbaum, P.H.
The measurement of meaning.
University of Illinois Press, Urbana, 1957.

[Palmer 86] Palmer, F.R.
Mood and Modality.
Cambridge University Press, Cambridge, England, 1986.
Cambridge Textbooks in Linguistics.

[Paris 87] Paris, C.L.
The Use of Explicit User models in Text Generation: Tailoring to a User's level of expertise.
PhD thesis, Columbia University, 1987.

[Patten 86] Patten, T.
Interpreting Systemic Grammar as a Computational Representation: A Problem Solving Approach to Text Generation.
PhD thesis, Edinburgh University, 1986.
Published by Cambridge University Press, 1988.

[Pereira 85] Pereira, F.
A Structure Sharing Formalism for Unification-based Formalisms.
In *Proceedings of the 23rd annual meeting of the ACL*, pages 137-144. ACL, 1985.

[Pereira 87] Pereira, F.C.N.
Grammars and Logics of Partial Information.
In *Proceedings of the International Conference on Logic Programming*. Melbourne, Australia, 1987.

[Polguere 90] Polguere, Alain.
Structuration et mise en jeu procedurale d'un modele linguistique declaratif dans un cadre de generation de texte.
PhD thesis, Universite de Montreal, 1990.

[Pollard & Sag 87]
Pollard, C. and I.A. Sag.
CSLI Lecture Notes. Volume 13: *Information-based Syntax and Semantics - Volume 1*.
University of Chicago Press, Chicago, Il, 1987.

[Prince 78] Prince, E.F.
A Comparison of Wh-Clefts and It-clefts in Discourse.
Language 54(4):883-906, December, 1978.

[Prince 81] Prince, E.F.
Toward a Taxonomy of Given-New Information.
In Cole, P. (editor), *Radical Pragmatics*, pages 223-256. Academic Press, New York, 1981.

[Quirk *et al* 72] Quirk, R. *et al*.
A Grammar of Contemporary English.
Longman, 1972.

[Raccah 87] Raccah, P.Y.
Modelling argumentation and modelling with argumentation.
Argumentation , 1987.

[Rambow & Korelsky 92]
Rambow, O. and Korelsky, T.
Applied Text Generation.
In *Proceedings of the 3rd Conference on Applied Natural Language Processing*, pages 40-47. Association for Computational Linguistics, Trento, Italy, April, 1992.

[Rayner and Banks 90]
Rayner, M. and A. Banks.
An implementable semantics for comparative constructions.
Computational Linguistics 16(2):86-112, June, 1990.

[Reichenbach 47] Reichenbach, H.
Elements of Symbolic Logic.
Macmillan, London, 1947.

[Reichman 85] Reichman, R.
Getting computers to talk like you and me: discourse context, focus and semantics (an ATN model).
MIT press, Cambridge, Ma, 1985.

[Reiter 90] Reiter, E.B.
Generating appropriate natural language object description.
PhD thesis, Center for research in computing technology, Harvard University, 1990.

[Reiter and Mellish 92]
Reiter, E. and C. Mellish.
Using Classification to Generate Text.
In *Proceedings of the 30th Meeting of the ACL*. University of Delaware, Newark, DE, 1992.

[Resnick et al. 90] Resnick, L.A. and Borgida, A. and Brachman, R.J. and McGuiness, D.L. and Patel-Schneider R.F.
CLASSIC: description and reference manual for the COMMON LISP implementation version 1.02
1990.

[Robin 90] Robin, J.
Lexical Choice in Natural Language Generation.
Technical Report CUCS-040-90, Columbia University, 1990.

[Robin 92a] Robin, J.
A Revision-based Architecture for Reporting Facts in their Historical Context.
In M. Zock and H. Horacek (editor), *Proceedings of the Third European Workshop on Language Generation (to appear)*. To appear, Judenstein, Austria, 1992.

[Robin 92b] Robin, J.
Generating Newswire Report Leads with Historical Information: a Draft and Revision Approach.
Technical Report , Columbia University, Dept. of Computer Science, New York, NY 10027, 1992.

[Roulet *et al* 85] Roulet, E. *et al*.
L'articulation du discours en francais contemporain.
Berne, Lang, 1985.

[Rubinoff 92] Rubinoff, R.
Negotiation, Feedback, and Perspective within Natural Language Generation.
PhD thesis, Dept of Computer and Information Sciences, University of Pennsylvannia, 1992.

[Rusiecki 85] Rusiecki, J.
Adjectives and Comparison in English: a semantic study.
Longman, England, 1985.
(Longman Linguistics Library).

[Sag and Pollard 91]
Sag, I. and C. Pollard.
An integrated theory of complement control.
Language 67(1):63-113, 1991.

[Sells 87] Sells, P.
Backward Anaphora and Discourse Structure: Some Considerations.
Technical Report CSLI-87-114, Center for the Study of Language and Information, November, 1987.

[Shieber 86] Shieber, S.
CSLI Lecture Notes. Volume 4: *An introduction to Unification-Based Approaches to Grammar*.
University of Chicago Press, Chicago, Il, 1986.

[Shieber 88] Shieber, S.M.
A uniform architecture for parsing and generation.
In *Proceedings of the 12th International Conference on Computational Linguistics (COLING'88)*, pages 614-619. Budapest, August 1988, 1988.

[Shieber et al 90] Shieber, S.M. and Van Noord, G. and Moore, R.M. and Pereira F.C.P.
Semantic Head-Driven Generation.
Computational Linguistics 16(1):30-42, 1990.

[Sidner 79] Sidner, C.L.
Towards a Computational Theory of Definite Anaphora Comprehension in English Discourse.
PhD thesis, MIT, 1979.

[Sinclair & Coulthard 75]
Sinclair and Coulthard.
Towards an Analysis of Discourse.
Oxford University Press, Oxford, England, 1975.

[Smadja 91a] Smadja, F.
Retrieving Collocational Knowledge from Textual Corpora. An Application: Language Generation..
PhD thesis, Computer Science Department, Columbia University, February, 1991.

[Smadja 91b] Smadja, F.
Microcoding the Lexicon with Co-Occurrence Knowledge.
In Uri Zernik (editor), *Lexical Acquisition: Using on-line resources to build a lexicon*. Lawrence Erlbaum, 1991.
In press.

[St-Dizier 92] Saint-Dizier, P.
A Constraint Logic Programming Treatment of Syntactic Choice in Natural Language Generation.
Lecture Notes in Artificial Intelligence 587. Aspects of Automated Natural Language Generation.
Springer-Verlag, 1992, pages 119-134.
Proceedings of the 6th International Workshop on Natural Language Generation, Trento, Italy.

[Talmy 76] Talmy, L.
Semantic causative types.
In Shibatani, M. (editor), *Syntax and semantics*. Volume 6: *The grammar of causative constructions*. Academic Press, London, 1976.

[Talmy 83] Talmy, L.
How language structures space.
In Pick, H.L., Acredolo, L.P. (editor), *Spatial orientation: theory, research and application*. Plenum Press, New York/London, 1983.

[Talmy 85] Talmy, L.
Lexicalization patterns: semantic structure in lexical form.
In Shopen, T. (editor), *Language typology and syntactic description*. Volume 3: *Grammatical categories and the lexicon*. Cambridge University Press, 1985.

[Taylor and Cameron 87]
Taylor, T.J. and Cameron, D.
Language & Communication Library. Volume 9: *Analysing Conversation: Rules and Units in the Structure of Talk.*
Pergamon Press, Oxford, 1987.

[Teyssier 68] Teyssier, J.
Notes on the syntax of the adjective in modern English.
Lingua 20:225-249, 1968.

[Uszkoreit 86] Uszkoreit, H.
Categorial Unification Grammars.
Technical Report CSLI-86-68, Center for the Study of Language and Information - Stanford University, 1986.

[van Benthem 89] van Benthem, J.
Polyadic quantifiers.
Linguistics and Philosophy 12:437-464, 1989.

[Van Noord 90] Van Noord, G.
An Overview of Head-driven Bottom-up Generation.
In Dale, R., Mellish, C. and Zock, M. (editor), *Current Research in Natural Language Generation*, pages 141-166. Academic Press, 1990.

[Vendler 68] Vendler, Zeno.
Adjectives and Nominalizations.
Mouton, The Hague, Paris, 1968.

[Webster 63] Merriam Webster.
Webster's Seventh New Collegiate Dictionary.
Merriam Webster, Springfield, MA, 1963.

[Wedekind 88] Wedekind, J.
Generation as structure driven derivation.
In *Proceedings of the 12th International Conference on Computational Linguistics (COLING'88)*, pages 732-737. Budapest, August 1988, 1988.

[Westerstahl 84] Westerstahl, D.
Determiners and Context Sets.
In van Benthem, J. and A. ter Meulen (editor), *Generalized Quantifiers*. Foris, Dordrecht, 1984.

[Williams 80] Williams, E.
Predication.
Linguistic Inquiry 11:203-238, 1980.

[Winograd 72] Winograd, T.
Understanding Natural Language.
Academic Press, New York, 1972.

[Winograd 83] Winograd, T.
Language as a Cognitive Process.
Addison-Wesley, Reading, Ma., 1983.

[Woods 70] Woods, W.A.
Transition network grammars for natural language analysis.
Communications of the ACM 13:591-606, 1970.

[Zadeh 84] Zadeh, L.A.
Precisiation of Meaning via Translation into PRUF.
In Vaina, L. and J. Hintikka (editor), *Cognitive Constraints on Communication*, pages 373-401.
D. Reidel Publishing Company, 1984.

Appendix A
SURGE Coverage: Clause Level

This appendix demonstrates the coverage of the part of the SURGE grammar dealing with clauses. As discussed in Sect.5.2 (p.127), there are tree types of decisions involved when processing a clause, which are encoded in three of SURGE's systems:

- **Transitivity system**: determines which configuration of participants is acceptable, and imposes a syntactic category on the realization of each participant.
- **Voice system**: maps the participants to grammatical functions like subject or object and imposes an order on the constituents. The voice system controls the choice of passive vs. active, but also choices like the dative shift, dislocations and clefts which affect the position of constituents in the clause by purely syntactic means.
- **Mood system**: determines if the clause is interrogative, imperative, assertive or non-finite (infinitive or participial).

This appendix lists input forms (in FD format) exercising most major forms of transitivity accepted by SURGE and illustrates the use of lexical process types to account for verbal patterns which are not captured by the standard transitivity patterns. In the following section, the voice system is demonstrated, with a list of inputs showing how passive can be selected and constraints on dative move. I then list inputs showing the mood system of the grammar. Inputs include all forms of declarative, interrogative and relative moods with examples of long distance dependencies. Non finite forms including imperative, infinitive, present and past participles and subjunctive are also illustrated. The next section shows coverage of modality and polarity, with examples of negation and of the different modals generated by the grammar. A list of circumstantials is then shown, with an example of each type. The tense system is finally shown, with a list of the 36 tenses at the active and passive form supported by SURGE. Finally, clause combining and ellipsis is demonstrated.

A.1. Transitivity System

In this section, all simple and composite transitivity clause patterns are tested. 36 distinct patterns are shown. Only active declarative voices are shown in this section. Variations on the clause structure, such as passive voice and dative move, are shown in Sect.A.3. For each pattern, features controlling the selection of the pattern are shown, for example `agentive` and `effective`. The transitivity system of SURGE is discussed in Sect.5.2.1 (p.127).

The following sentences are shown for the tests:

Bo runs.
Bo drinks protein shakes.
Bo cooks dinner.
Bo takes a walk.
Bo climbs the mountain.
The window opens.
The window pops.
Bo knows.
Bo knows FUF.
Bo is strong.
Bo is the greatest.

Bo has sneakers.
Bo owns sneakers.
Bo is here.
Bo occupies the left flank.
The race is now.
The steeplechase follows the high jump.
Bo is with Mo.
Bo accompanies Mo.
There is a bug.
It rains.
Nike made Bo rich.
The deal makes Bo the richest.
Bo gave the Raiders the victory.
Bo gave the victory to the Raiders.
Bo lifted the Raiders to a victory.
It pops a window on the screen.
Bo got tough.
Bo became the best.
Bo bought sneakers.
Bo ran home.
Bo grew old.
Bo received the ball.
Bo fell down.
The window popped wide.
The window popped on the screen.

For each test, the sentence is shown along with the input FD producing the sentence when unified with SURGE. In the examples, two-letter abbreviations are used in comments to describe the configuration of roles corresponding to each transitivity class. The abbreviations are explained in Fig.A-1.

Ag: Agent
Af: Affected
Cr: Created
Rg: Range
Ca: Carrier
At: Attribute
Id: Identified
Ir: Identifier
Pr: Processor
Ph: Phenomenon
Pr: Possessor
Pd: Possessed
Ld: Located
Ln: Location
Tm: Time

Figure A-1: List of abbreviations for names of roles in Surge

```
;; ================================================================
;; test of clause patterns of the transitvity system:
;; ================================================================
;; simple process
;; event
;; material
;; agentive non-effective: Ag


;; Store-verbs is used to inform the morphology module
;; of irregular verbs.
(store-verbs '( ("run" "runs" "ran" "running" "run")) )
(def-test t100
  "Bo runs."
  ((cat clause)
   (proc ((type material)
          (effective no)
          ;;         (agentive yes)  is default
          (lex "run")))
   (partic ((agent ((cat proper) (lex "Bo")))))))



;; agentive dispositive: Ag + Af
(def-test t101
  "Bo drinks protein shakes."
  ((cat clause)
   (proc ((type material)
          (lex "drink")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (affected ((cat common)
                       (number plural)
                       (definite no)
                       (classifier === "protein")
                       (head === "shake")))))))



;; agentive creative: Ag + Cr
;; Note the zero-determiner used for dinner (cf. App.B for details).
(def-test t102
  "Bo cooks dinner."
  ((cat clause)
   (proc ((type material) (effect-type creative) (lex "cook")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (created ((cat common) (lex "dinner")
                      (denotation meal)))))))



;; agentive with range: Ag + Rg
(def-test t103
  "Bo takes a walk."
  ((cat clause)
   (proc ((type material)
          (effective no)
          (event-as participant)
          (lex "take")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (range ((cat common) (definite no) (lex "walk")))))))



(def-test t104
  "Bo climbs the mountain."
  ((cat clause)
   (proc ((type material)
          (effective no)
```

```
          (event-as process)
          (lex "climb")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (range ((cat common) (lex "mountain")))))))


;; non-agentive dispositive: Af
(def-test t105
  "The window opens."
  ((cat clause)
   (proc ((type material) (agentive no) (lex "open")))
   (partic ((affected ((cat common) (lex "window")))))))


;; non-agentive creative: Cr
(def-test t106
  "The window pops."
  ((cat clause)
   (proc ((type material)
          (agentive no)
          (effect-type creative)
          (lex "pop")))
   (partic ((created ((cat common) (lex "window")))))))


;; mental non-transitive: Pr
(def-test t107
  "Bo knows."
  ((cat clause)
   (proc ((type mental) (transitive no) (lex "know")))
   (partic ((processor ((cat proper) (lex "Bo")))))))


;; mental transitive: Pr + Ph
(def-test t108
  "Bo knows FUF."
  ((cat clause)
   (proc ((type mental)
          (lex "know")))
   (partic ((processor ((cat proper) (lex "Bo")))
            (phenomenon ((cat proper) (lex "FUF")))))))


;; relation
;; ascriptive attributive: Ca + At
;; In most of the relation examples, a default verb is selected by the
;; grammar itself (have, own, be...)

(def-test t109
  "Bo is strong."
  ((cat clause)
   (proc ((type ascriptive)
          (mode attributive)))
   (partic ((carrier ((cat proper) (lex "Bo")))
            (attribute ((cat ap) (lex "strong")))))))



;; ascriptive equative: Id + Ir
(def-test t110
  "Bo is the greatest."
  ((cat clause)
```

```
   (proc ((type ascriptive) (mode equative)))
   (partic ((identified ((cat proper) (lex "Bo")))
            (identifier ((cat common) (lex "greatest")))))))


;; possessive attributive: Pr + Pd
(def-test t111
  "Bo has sneakers."
  ((cat clause)
   (proc ((type possessive)))
   (partic ((possessor ((cat proper) (lex "Bo")))
            (possessed ((cat common)
                        (number plural)
                        (definite no)
                        (lex "sneaker")))))))


;; possesive equative: Id + Ir
(def-test t112
  "Bo owns sneakers."
  ((cat clause)
   (proc ((type possessive) (mode equative)))
   (partic ((identified ((cat proper) (lex "Bo")))
            (identifier ((cat common)
                         (number plural)
                         (definite no)
                         (lex "sneaker")))))))


;; locative spatial atributive: Ld + Ln
(def-test t113
  "Bo is here."
  ((cat clause)
   (proc ((type spatial)))
   (partic ((located ((cat proper) (lex "Bo")))
            (location ((cat adv) (lex "here")))))))


;; locative spatial equative: Id + Ir
;; Equative mode is used when the process is reversible (passive can be
;; used: the left flank is occupied by Bo).
(def-test t114
  "Bo occupies the left flank."
  ((cat clause)
   (proc ((type spatial) (mode equative) (lex "occupy")))
   (partic ((located ((cat proper) (lex "Bo")))
            (location ((cat common)
                       (classifier === "left")
                       (head === "flank")))))))


;; locative temporal attributive: Ld + Tm
(def-test t115
  "The race is now."
  ((cat clause)
   (proc ((type temporal)))
   (partic ((located ((cat common) (lex "race")))
            (time ((cat adv) (lex "now")))))))


;; locative spatial equative: Id + Ir
(def-test t116
```

```
  "The steeplechase follows the high jump."
  ((cat clause)
   (proc ((type temporal) (mode equative) (lex "follow")))
   (partic ((identified ((cat common) (lex "steeplechase")))
            (identifier ((cat common)
                         (classifier === "high")
                         (head === "jump")))))))


;; locative accompaniment (always equative): Ld/Id + Ac/Ir
(def-test t117
  "Bo is with Mo."
  ((cat clause)
   (proc ((type accompaniment)))
   (partic ((located ((cat proper) (lex "Bo")))
            (location ((cat pp)
                       (prep === with)
                       (np ((cat proper) (lex "Mo")))))))))


(def-test t118
  "Bo accompanies Mo."
  ((cat clause)
   (proc ((type accompaniment) (mode equative) (lex "accompany")))
   (partic ((identified ((cat proper) (lex "Bo")))
            (identifier ((cat proper) (lex "Mo")))))))


;; locative existential: Ld
(def-test t119
  "There is a bug."
  ((cat clause)
   (proc ((type existential)))
   (partic ((located ((cat common) (definite no) (lex "bug")))))))


;; locative natural-phenom: no participant
(def-test t120
  "It rains."
  ((cat clause) (proc ((type natural-phenom) (lex "rain")))))



;; ============================================================
;; COMPOSITE PROCESSES
;; ============================================================

;; agentive dispositive ascriptive attributive: Ag + Af/Ca + At
(def-test t121
  "Nike made Bo rich."
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (relation-type ascriptive)
          ;;           (agentive yes)         is default
          ;;           (effective yes)        is default
          ;;           (effect-type affected) is default
          ;;           (mode attributive)     is default
          (lex "make")))
   (partic ((agent ((cat proper) (lex "Nike")))
            (affected ((cat proper) (lex "Bo")))
            (carrier {^ affected})
            (attribute ((cat ap) (lex "rich")))))))
```

```
;; agentive dispositive ascriptive equative: Ag + Af/Id + At
(def-test t122
  "The deal makes Bo the richest."
  ((cat clause)
   (proc ((type composite)
          (relation-type ascriptive)
          (mode equative)
          (lex "make")))
   (partic ((agent ((cat common) (lex "deal")))
            (affected ((cat proper) (lex "Bo")))
            (identified {^ affected})
            (identifier ((cat common) (lex "richest")))))))


;; agentive dispositive possessive attributive: Ag + Af/Pr + Pd
(def-test t123
  "Bo gave the Raiders the victory."
  ((cat clause)
   (tense past)
   (proc ((type composite) (relation-type possessive) (lex "give")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (affected ((cat proper) (lex "Raider") (number plural)))
            (possessor {^ affected})
            (possessed ((cat common) (lex "victory")))))))


;; agentive dispositive locative attributive: Ag + Af/Ld + Ln
(def-test t124
  "Bo lifted the Raiders to a victory."
  ((cat clause)
   (tense past)
   (proc ((type composite) (relation-type locative) (lex "lift")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (affected ((cat proper) (lex "Raider") (number plural)))
            (located {^ affected})
            (location ((cat pp)
                       (prep ((lex "to")))
                       (np ((cat common)
                            (definite no)
                            (lex "victory")))))))))


;; agentive creative locative attributive: Ag + Cr/Ld + Ln
(def-test t125
  "It pops a window on the screen."
  ((cat clause)
   (proc ((type composite)
          (relation-type locative)
          (effect-type creative)
          (lex "pop")))
   (partic ((agent ((cat personal-pronoun)
                    (person third)
                    (gender neuter)
                    (number singular)))
            (created ((cat common) (definite no) (lex "window")))
            (located {^ created})
            (location ((cat pp)
                       (prep ((lex "on")))
                       (np ((cat common) (lex "screen")))))))))


;; agentive non-effective ascriptive attributive: Ag/Ca + At
(def-test t126
  "Bo got tough."
```

```
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (relation-type ascriptive)
          (effective no)
          (lex "get")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (carrier {^ agent})
            (attribute ((cat ap) (lex "tough")))))))


;; agentive non-effective ascriptive equative: Ag/Id + Ir
(def-test t127
  "Bo became the best."
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (relation-type ascriptive)
          (mode equative)
          (effective no)
          (lex "become")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (identified {^ agent})
            (identifier ((cat common) (lex "best")))))))


;; agentive non-effective possessive attributive: Ag/Pr + Pd
(def-test t128
  "Bo bought sneakers."
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (relation-type possessive)
          (effective no)
          (lex "buy")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (possessor {^ agent})
            (possessed ((cat common)
                        (definite no)
                        (number plural)
                        (lex "sneaker")))))))


;; agentive non-effective locative attributive: Ag/Ld + Ln
(def-test t129
  "Bo ran home."
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (relation-type locative)
          (effective no)
          (lex "run")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (located {^ agent})
            (location ((cat adv) (lex "home")))))))


;; non-agentive dispositive ascriptive attributive: Af/Ca + At
(def-test t130
  "Bo grew old."
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (agentive no)
```

```
          (relation-type ascriptive)
          (lex "grow")))
   (partic ((affected ((cat proper) (lex "Bo")))
            (carrier {^ affected})
            (attribute ((cat ap) (lex "old")))))))


;; non-agentive dispositive possessive attributive: Af/Pr + Pd
(def-test t131
  "Bo received the ball."
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (agentive no)
          (relation-type possessive)
          (lex "receive")))
   (partic ((affected ((cat proper) (lex "Bo")))
            (possessor {^ affected})
            (possessed ((cat common) (lex "ball")))))))


;; non-agentive dispositive locative attributive: Af/Ld + Ln
(def-test t132
  "Bo fell down."
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (agentive no)
          (relation-type locative)
          (lex "fall")))
   (partic ((affected ((cat proper) (lex "Bo")))
            (located {^ affected})
            (location ((cat adv) (lex "down")))))))


;; non-agentive creative asciptive attributive: Cr/Ca + At
(def-test t133
  "The window popped wide."
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (agentive no)
          (effect-type creative)
          (relation-type ascriptive)
          (lex "pop")))
   (partic ((created ((cat common) (lex "window")))
            (carrier {^ created})
            (attribute ((cat ap) (lex "wide")))))))


;; non-agentive creative locative attributive: Cr/Ld + Ln
(def-test t134
  "The window popped on the screen."
  ((cat clause)
   (tense past)
   (proc ((type composite)
          (agentive no)
          (effect-type creative)
          (relation-type locative)
          (lex "pop")))
   (partic ((created ((cat common) (lex "window")))
            (located {^ created})
            (location ((cat pp)
                       (prep ((lex "on")))
```

```
(np ((cat common) (lex "screen")))))))))
```

A.2. Lexical Processes and Control

This section shows how the transitivity system can be avoided when the lexicon directly provides information about the subcategorization frame of the verb. The lexical process type is then used. Lexical process types are used in two cases: (1) if the meaning of the verb does not fit well under the semantic classes defined in the transitivity system (*e.g.*, relation, material action etc); and (2) if the verb has syntactic properties which are not accounted for by the transitivity system. In the following examples, the control behavior of the verbs justifies the selection of a lexical process type. The last example is the most complex of all. It is a case of syntactic inference discussed in Sect.5.1.9 (p.126).

The examples listed in this section are:

The customer persuaded the programmer that there is a bug.
The customer persuaded the programmer to revise the code.
The customer wants him to do it.
Migraine abortive treatment requires you to take a drug at the immediate onset of headaches.
SURGE requires the input to be lexicalized by the lexical chooser.

```
;; ---------------------------------------------------------------------
;; TEST LEXICAL ROLES AND CONTROL
;; ---------------------------------------------------------------------

(def-test t140
  "The customer persuaded the programmer that there is a bug."
  ((cat clause)
   (tense past)
   (proc ((type lexical)
          (lex "persuade")
          (subcat ((1 {^3 lex-roles persuader})
                   (2 {^3 lex-roles persuadee})
                   (3 {^3 lex-roles soa})
                   (1 ((cat np)))
                   (2 ((cat np)))
                   (3 ((cat clause)
                       (mood bound)
                       (binder === that)))))))
   (lex-roles ((persuader === customer)
               (persuadee === programmer)
               (soa ((proc ((type existential)))
                     (partic ((located ((cat common)
                                        (definite no)
                                        (lex "bug")))))))))))



(def-test t141
  "The customer persuaded the programmer to revise the code."
  ((cat clause)
   (tense past)
   (proc ((type lexical)
          (lex "persuade")
          (subcat ((1 {^3 lex-roles persuader})
                   (2 {^3 lex-roles persuadee})
                   (3 {^3 lex-roles soa})
                   (1 ((cat np)))
                   (2 ((cat np)))
                   (3 ((cat clause)
                       (mood infinitive)
                       (oblique ((1 ((index {^4 2 index})
                                     (gap yes)))))))))))
   (lex-roles
    ((persuader === customer)
     (persuadee === programmer)
     (soa ((proc ((type material)
                  (lex "revise")))
            (partic ((affected === code)))))))))



;; With a more general approach ala HPSG [Sag and Pollard 91]
;; Roles influence/influencer/soa (state of affairs)
(def-test t142
  "The customer wants him to do it."
  ((cat clause)
   (tense present)
   (proc ((type lexical)
          (lex "want")
          (voice active)
          (subcat ((1 {^3 lex-roles influence})
                   (2 {^3 lex-roles influenced})
                   (3 {^3 lex-roles soa})
                   (3 ((cat clause)
                       (mood infinitive)
                       (oblique ((1 ((index {^4 2 index}) ; controlled
```

```
                                    (gap yes)))))))))))
   (lex-roles
    ((influence === customer)
     (influenced ((cat personal-pronoun)
                  (person third)
                  (gender masculine)
                  (number singular)))
     (soa ((proc ((type material)
                  (lex "do")))
           (partic ((affected ((cat personal-pronoun)
                               (person third)
                               (gender neuter)
                               (number singular)))))))))


;; Another lexical process with the same subcat frame: require.
(def-test t215
  "Migraine abortive treatment requires you to take a drug at the
   immediate onset of headaches."
  ((cat clause)
   (process ((lex "require")
             (subcat ((1 {^3 lex-roles influence})
                      (2 {^3 lex-roles influenced})
                      (3 {^3 lex-roles soa})
                      (1 ((cat np)))
                      (2 ((cat np)))
                      (3 ((cat clause)
                          (mood infinitive)
                          (oblique ((1 ((index {^4 2 index})
                                        (gap yes)))))))))
             (type lexical)))
   (lex-roles ((influence
                ((lex "migraine abortive treatment") (cat proper)))
               (influenced ((cat personal-pronoun) (person second)))
               (soa ((circum
                      ((time ((time-type "at")(cat np)
                              (describer ((lex "immediate")))
                              (lex "onset")
                              (qualifier ((cat pp)
                                          (np ((lex "headache")
                                               (definite no)
                                               (number plural)))))
                              (definite yes)))))
                     (cat clause)
                     (process-type material)
                     (process ((lex "take")))
                     (partic ((affected ((lex "drug") (definite no)
                                         (cat common)))))))))))


;; Example of syntactic inference: interaction control/passive
;; selection. Because of the coindexing soa-affected/influenced,
;; the passive voice is selected in the soa-clause so that affected
;; becomes the subject (which is the controlled element).
(def-test t216
  "SURGE requires the input to be lexicalized by the lexical chooser."
  ((cat clause)
   (process ((lex "require")
             (subcat ((1 {^3 lex-roles influence})
                      (2 {^3 lex-roles influenced})
                      (3 {^3 lex-roles soa})
                      (1 ((cat np)))
                      (2 ((cat np)))
                      (3 ((cat clause)
                          (mood infinitive)
                          (oblique ((1 ((index {^4 2 index})
```

```
                                        (gap yes)))))))))
          (type lexical)))
(lex-roles ((influence
             ((lex "SURGE") (cat proper)))
            (influenced ((cat common) (lex "input")
                         (index ((concept input1)))
                         (definite yes)))
            (soa ((process-type material)
                  (process ((lex "lexicalize")))
                  ;; Affected co-indexed with influenced!
                  (partic ((affected ((index ((concept input1)))))
                           (agent ((cat common)
                                   (lex "lexical chooser")
                                   (definite yes)))))))))))
```

A.3. Voice System

After the transitivity system has mapped semantic roles to an oblique frame, the voice system maps the oblique frame to syntactic functions such as subject, object, iobject etc. Two effects are currently covered in SURGE: passivation and dative move. There are constraints on when each one is selected. Various cases are illustrated in the following examples:

A science book is given by John to Mary.
A science book is given to Mary.
Old McDonald had a farm.
Old McDonald owned a farm.
A farm was owned by Old McDonald.
SURGE requires the input to be lexicalized by the lexical chooser.
Mary is given a science book by John.
A science book is given to Mary by John.
Bo gave the Raiders the victory.
Bo gave the victory to the Raiders.
John gives a blue book to Mary.
John gives it to Mary.
A science book is given by John to Mary.
The person to whom John will give a blue book.
Who gives a blue book to Mary?
What will John give to Mary?
To whom will John give a blue book?
The person by whom a blue book will be given to John.

```
;; ============================================================
;; Passivation constraints
;; ============================================================

;; Simple passivation
(def-test t3
  "A science book is given by John to Mary."
  ((cat clause)
   (proc ((type composite)
          (relation-type possessive)
          (voice passive)
          (lex "give")
          (dative-prep "to")))
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessed ((cat common)
                        (lex "book")
                        (definite no)
                        (classifier === "science")))))))


;; Simple passivation - agentless (even if agent is provided)
(def-test t3bis
  "A science book is given to Mary."
  ((cat clause)
   (proc ((type composite)
          (relation-type possessive)
          (voice passive)
          (lex "give")
          (dative-prep "to")))
   (agentless yes)
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessed ((cat common)
                        (lex "book")
                        (definite no)
                        (classifier === "science")))))))


;; Passivation of equative relations:
;; have (not passivable) vs. own (passivable)
(def-test t48
  "Old McDonald had a farm."
  ((cat clause)
   (proc ((type possessive) (mode attributive)))
   (tense past)
   (partic ((possessor ((cat proper) (head === "Old McDonald")))
            (possessed ((cat common) (head === farm)
                        (definite no)))))))



(def-test t49
  "Old McDonald owned a farm."
  ((cat clause)
   (proc ((type possessive) (mode equative)))
   (tense past)
   (partic ((possessor ((cat proper) (head === "Old McDonald")))
            (possessed ((cat common) (head === farm)
                        (definite no)))))))



(def-test t50
  "A farm was owned by Old McDonald."
  ((cat clause)
   (proc ((type possessive) (mode equative) (voice passive)))
```

```
   (tense past)
   (agentless no)
   (partic ((possessor ((cat proper) (head === "Old McDonald")))
            (possessed ((cat common) (head === farm)
                        (definite no)))))))


;; Because of the coindexing soa-affected/influenced,
;; the passive voice is selected in the soa-clause
;; (so that affected becomes the subject).
(def-test t216
  "SURGE requires the input to be lexicalized by the lexical chooser."
  ((cat clause)
   (process ((lex "require")
             (subcat ((1 {^3 lex-roles influence})
                      (2 {^3 lex-roles influenced})
                      (3 {^3 lex-roles soa})
                      (1 ((cat np)))
                      (2 ((cat np)))
                      (3 ((cat clause)
                          (mood infinitive)
                          (oblique ((1 ((index {^4 2 index})
                                        (gap yes)))))))))
             (type lexical)))
   (lex-roles ((influence
                ((lex "SURGE") (cat proper)))
               (influenced ((cat common) (lex "input")
                            (index ((concept input1)))
                            (definite yes)))
               (soa ((process-type material)
                     (process ((lex "lexicalize")))
                     ;; Affected co-indexed with influenced!
                     (partic ((affected ((index ((concept input1)))))
                              (agent ((cat common)
                                      (lex "lexical chooser")
                                      (definite yes)))))))))))


;; Selection of passive by focus constraint:
;; focus tends to become subject when possible.
;; Simple passivation
(def-test t3-ter
  "Mary is given a science book by John."
  ((cat clause)
   (focus {partic affected})
   (proc ((type composite)
          (relation-type possessive)
          (voice passive)
          (lex "give")
          (dative-prep "to")))
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessed ((cat common)
                        (lex "book")
                        (definite no)
                        (classifier === "science")))))))


(def-test t3-quad
  "A science book is given to Mary by John."
  ((cat clause)
   (focus {partic possessed})
   (proc ((type composite)
          (relation-type possessive)
          (voice passive)
          (lex "give")
```

```
          (dative-prep "to")))
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessed ((cat common)
                        (lex "book")
                        (definite no)
                        (classifier === "science")))))))


;; ============================================================
;; Dative move constraints
;; ============================================================

;; Simple cases: grammar is free to choose either dative move or not.
;; agentive dispositive possessive attributive: Ag + Af/Pr + Pd
(def-test t123
  "Bo gave the Raiders the victory."
  ((cat clause)
   (tense past)
   (proc ((type composite) (relation-type possessive) (lex "give")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (affected ((cat proper) (lex "Raider") (number plural)))
            (possessor {^ affected})
            (possessed ((cat common) (lex "victory")))))))


;; agentive dispositive possessive attributive: Ag + Af/Pr + Pd
(def-test t123-bis
  "Bo gave the victory to the Raiders."
  ((cat clause)
   (tense past)
   (dative-move no)
   (proc ((type composite) (relation-type possessive) (lex "give")))
   (partic ((agent ((cat proper) (lex "Bo")))
            (affected ((cat proper) (lex "Raider") (number plural)))
            (possessor {^ affected})
            (possessed ((cat common) (lex "victory")))))))


(def-test t2
  "John gives a blue book to Mary."
  ((cat clause)
   (proc ((type composite)
          (relation-type possessive)
          (lex "give")))
   (dative-move no)
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessor {^ affected})
            (possessed ((lex "book")
                        (cat common)
                        (definite no)
                        (describer === "blue")))))))


;; ------------------------------------------------------------
:; In all the following cases, dative move is inferred by the grammar
;; because of the interaction with another syntactic decision.
;; ------------------------------------------------------------

;; Interaction dative move with pronouns:
;; fail with (dative-move yes)
(def-test t2bis
  "John gives it to Mary."
  ((cat clause)
   (proc ((type composite)
          (relation-type possessive)
          (lex "give")))
```

```
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessor {^ affected})
            (possessed ((cat pronoun)))))))


;; Interaction dative move with passive:
;; fail with (dative-move yes)
(def-test t3
  "A science book is given by John to Mary."
  ((cat clause)
   (proc ((type composite)
          (relation-type possessive)
          (voice passive)
          (lex "give")
          (dative-prep "to")))
   (agentless no)
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessed ((cat common)
                        (lex "book")
                        (definite no)
                        (classifier === science)))))))


;; Interaction dative move with relative mood:
;; would fail with (dative-move yes)
(def-test t97
  "The person to whom John will give a blue book."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (tense future)
               (scope {^ partic possessor})
               (proc ((type composite)
                      (relation-type possessive)
                      (lex "give")))
               (partic ((agent ((lex "John") (cat proper)))
                        (possessed ((lex "book")
                                    (cat common)
                                    (definite no)
                                    (describer === "blue")))))))))


;; Interaction dative move with interrogative mood:
;; Would fail with (dative-move yes)
(def-test t71
  "Who gives a blue book to Mary?"
  ((cat clause)
   (mood wh)
   (scope {^ partic agent})
   (proc ((type composite)
          (relation-type possessive)
          (lex "give")))
   (partic ((agent ((animate yes)))
            (possessed ((lex "book")
                        (cat common)
                        (definite no)
                        (describer === "blue")))
            (affected ((lex "Mary") (cat proper)))
            (possessor {^ affected})))))



;; Interaction dative move with interrogative mood:
;; Would fail with (dative-move yes)
(def-test t72
```

```
  "What will John give to Mary?"
  ((cat clause)
   (mood wh)
   (tense future)
   (scope {^ partic possessed})
   (proc ((type composite)
          (relation-type possessive)
          (dative-prep "to")
          (lex "give")))
   (partic ((agent ((lex "John") (cat proper)))
            (affected ((lex "Mary") (cat proper)))
            (possessor {^ affected})))))


;; Interaction dative move with interrogative mood:
;; Would fail with (dative-move yes)
(def-test t73
  "To whom will John give a blue book?"
  ((cat clause)
   (mood wh)
   (tense future)
   (scope {^ partic possessor})
   (proc ((type composite)
          (relation-type possessive)
          (lex "give")))
   (partic ((agent ((lex "John") (cat proper)))
            (possessor ((animate yes)))
            (possessed ((lex "book")
                        (cat common)
                        (definite no)
                        (describer === "blue")))))))


;; Limitation of the grammar: can't currently generate:
;; The person by whom John will be given a blue book.
;; due to treatment of dative-move always no when displaced constituent
;; is given.
;; This makes life much easier and is not such a restriction.
(def-test t97bis
  "The person by whom a blue book will be given to John."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (tense future)
               (scope {^ partic agent})
               (proc ((type composite)
                      (relation-type possessive)
                      (voice passive)
                      (lex "give")))
               (partic ((possessor ((lex "John") (cat proper)))
                        (possessed ((lex "book")
                                    (cat common) (definite no)
                                    (describer === blue)))))))))
```

A.4. Mood

The mood system determines whether the clause is assertive, interrogative, imperative or relative. The following moods are used in SURGE:

```
declarative                  AI has six assignments.
interrogative yes-no         Does AI have six assignments?
interrogative wh             Who teaches AI?
bound                        whether AI has six assignments.
simple-relative              the person who teaches AI.
embedded-relative            the topics with which AI deals.
be-deleted relative          the topics covered in AI.
wh-nominal relative          I know who teaches AI.
wh-ever-nominal relative     whoever teaches AI.
imperative                   Take AI.
present-participle           the person taking AI.
infinitive                   for him to take AI.
past-participle              the topics taught in AI.
subjunctive                  that AI be taught.
```

In general, all embedded moods are preselected during clause planning, *i.e.*, when a clause is an argument of a matrix clause, its mood is determined by the main verb of the matrix clause. The following examples are shown in this section:

This car is expensive.
John did give Mary a blue book.
Is this car expensive?
Will John give Mary a blue book?
What is expensive?
How is this car?
What is a classic?
Who is your father?
Which is your father?
Where is your mother?
Who is in your house?
What covers the opening?
What does the seal cover?
When is the game?
What happens then?
The game happened then.
How is your sister?
With whom is your sister?
Who has a PhD?
What does she have?
Who owns this book?
Which does she own?
Who gives a blue book to Mary?
What will John give to Mary?
To whom will John give a blue book?
What does Deborah prefer?
Who worships Baal?
Why does he do it?
From where does the SIG-display marker move to the right?
Who do you think won the prize?
You think that John won the prize.
You think that the prize was won by John.
Do you think that John won the prize?
The customer persuaded the programmer that there is a bug.
The box which is expensive.
The box which is a classic.
The man who is your father.

The man that your father is.
The house where your mother is.
The man who is in your house.
The plate which covers the opening.
The hole which the seal covers.
The time when the game starts.
The thing that happens then.
The position from which the SIG-display marker moves to the right.
The way which your sister is.
The person with whom your sister is.
The person who has a PhD.
The box which she has.
The person who owns this book.
The person by whom this book is owned.
The box that she owns.
The person who gives a blue book to Mary.
The box which John will give to Mary.
The person to whom John will give a blue book.
The person by whom a blue book will be given to John.
The box which Deborah prefers.
The person who worships Baal.
The person who you think won the prize.
The person by whom you think the prize was won.
Take the hammer and hit the nail.
the power level’s increasing causes the sig-display marker to move to the right.
Karen demanded that SURGE be used in 1992.

```
;; ============================================================
;; Finite mood
;; ============================================================

;; Declarative
(def-test t1
  "This car is expensive."
  ((cat clause)
   (proc ((type ascriptive)))
   (partic ((carrier ((lex "car") (cat common) (distance near)))
            (attribute ((cat ap) (lex "expensive")))))))


;; Declarative with insistence
(def-test t54
  "John did give Mary a blue book."
  ((cat clause)
   (insistence yes)
   (proc ((type composite)
          (relation-type possessive)
          (lex "give")))
   (tense past)
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessor {^ affected})
            (possessed ((lex "book") (cat common) (definite no)
                        (describer === "blue")))))))


;; Yes-no
(def-test t52
  "Is this car expensive?"
  ((cat clause)
   (mood yes-no)
   (proc ((type ascriptive)))
   (partic ((carrier ((lex "car") (cat common) (distance near)))
            (attribute === "expensive")))))



(def-test t53
  "Will John give Mary a blue book?"
  ((cat clause)
   (mood yes-no)
   (tense future)
   (proc ((type composite)
          (relation-type possessive)
          (lex "give")))
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessor {^ affected})
            (possessed ((lex "book")
                        (cat common)
                        (definite no)
                        (describer === "blue")))))))



;; ------------------------------------------------------------------
;; WH interrogative
;; ------------------------------------------------------------------
;; Note how the scope of the question is specified
;; and how the grammar selects the appropriate question pronoun.

(def-test t55
  "What is expensive?"
  ((cat clause)
   (mood wh)
```

```
   (scope {^ partic carrier})
   (proc ((type ascriptive)))
   (partic ((attribute === "expensive")))))


(def-test t55bis
  "How is this car?"
  ((cat clause)
   (mood wh)
   (scope {^ partic attribute})
   (proc ((type ascriptive)))
   (partic ((attribute === "expensive")
            (carrier ((lex "car")
                      (cat common)
                      (distance near)))))))


(def-test t56
  "What is a classic?"
  ((cat clause)
   (mood wh)
   (proc ((type ascriptive)))
   (scope {^ partic carrier})
   (partic ((attribute ((cat common) (definite no)
                        (lex "classic")))))))



(def-test t57
  "Who is your father?"
  ((cat clause)
   (mood wh)
   (scope {^ partic identified})
   (proc ((type ascriptive) (mode equative)))
   (partic ((identified ((animate yes)))
            (identifier ((cat common)
                         (possessor ((cat personal-pronoun)
                                     (person second)))
                         (head === "father")))))))



;; "Which" selected in the case of equative:
;; pick out an element out of a set.
(def-test t58
  "Which is your father?"
  ((cat clause)
   (mood wh)
   (scope {^ partic identifier})
   (proc ((type ascriptive) (mode equative)))
   (partic ((identified ((cat common)
                         (possessor ((cat personal-pronoun)
                                     (person second)))
                         (head === "father")))))))



(def-test t59
  "Where is your mother?"
  ((cat clause)
   (mood wh)
   (scope {^ partic location})
   (proc ((type locative)))
   (partic ((located ((cat common)
                      (possessor ((cat personal-pronoun)
                                  (person second)))
                      (head === "mother")))))))
```

```
(def-test t60
  "Who is in your house?"
  ((cat clause)
   (mood wh)
   (scope {^ partic located})
   (proc ((type locative)))
   (partic ((located ((animate yes)))
            (location ((cat pp)
                       (prep === "in")
                       (np ((cat common)
                            (possessor ((cat personal-pronoun)
                                        (person second)))
                            (head === "house")))))))))


(def-test t61
  "What covers the opening?"
  ((cat clause)
   (mood wh)
   (scope {^ partic located})
   (proc ((type locative) (mode equative) (lex "cover")))
   (partic ((location ((cat common) (head === "opening")))))))


(def-test t62
  "What does the seal cover?"
  ((cat clause)
   (mood wh)
   (scope {^ partic location})
   (proc ((type locative) (mode equative) (lex "cover")))
   (partic ((located ((cat common) (head === "seal")))))))


(def-test t63
  "When is the game?"
  ((cat clause)
   (mood wh)
   (scope {^ partic time})
   (proc ((type temporal)))
   (partic ((located ((cat common) (head === "game")))))))



(def-test t64
  "What happens then?"
  ((cat clause)
   (mood wh)
   (scope {^ partic located})
   (proc ((type temporal) (lex "happen")))
   (partic ((time ((cat adv) (lex "then")))))))


(store-verbs
 '( ("happen" "happens" "happened" "happening" "happened")) )
(def-test t64bis
  "The game happened then."
  ((cat clause)
   (tense past)
   (proc ((type temporal) (lex "happen")))
   (partic ((located ((cat np) (head === game)))
            (time ((cat adv) (lex "then")))))))


(def-test t65
```

```
  "How is your sister?"
  ((cat clause)
   (mood wh)
   (scope {^ partic attribute})
   (proc ((type ascriptive)))
   (partic ((carrier ((cat common)
                      (possessor ((cat personal-pronoun)
                                  (person second)))
                      (head === "sister")))))))


(def-test t66
  "With whom is your sister?"
  ((cat clause)
   (mood wh)
   (proc ((type accompaniment)))
   (scope {^ partic location})
   (partic ((location ((animate yes)))
            (located ((cat common)
                      (possessor ((cat personal-pronoun)
                                  (person second)))
                      (head === "sister")))))))


(def-test t67
  "Who has a PhD?"
  ((cat clause)
   (mood wh)
   (scope {^ partic possessor})
   (proc ((type possessive)))
   (partic ((possessor ((animate yes)))
            (possessed ((cat common) (definite no)
                        (head === "PhD")))))))


(def-test t68
  "What does she have?"
  ((cat clause)
   (mood wh)
   (proc ((type possessive)))
   (scope {^ partic possessed})
   (partic ((possessor ((cat personal-pronoun)
                        (gender feminine)
                        (number singular)
                        (person third)))))))


(def-test t69
  "Who owns this book?"
  ((cat clause)
   (mood wh)
   (proc ((type possessive) (mode equative)))
   (scope {^ partic possessor})
   (partic ((possessor ((animate yes)))
            (possessed ((cat common)
                        (distance near)
                        (head === "book")))))))


(def-test t70
  "Which does she own?"
  ((cat clause)
   (mood wh)
   (proc ((type possessive) (mode equative)))
```

```
   (scope {^ partic possessed})
   (partic ((possessed ((restrictive yes)))    ;; which vs. what.
            (possessor ((cat personal-pronoun)
                        (gender feminine)
                        (person third)
                        (number singular)))))))


(def-test t71
  "Who gives a blue book to Mary?"
  ((cat clause)
   (mood wh)
   (scope {^ partic agent})
   (proc ((type composite)
          (relation-type possessive)
          (lex "give")))
   (partic ((agent ((animate yes)))
            (possessed ((lex "book")
                        (cat common)
                        (definite no)
                        (describer === "blue")))
            (affected ((lex "Mary") (cat proper)))
            (possessor {^ affected})))))


(def-test t72
  "What will John give to Mary?"
  ((cat clause)
   (mood wh)
   (tense future)
   (scope {^ partic possessed})
   (proc ((type composite)
          (relation-type possessive)
          (dative-prep "to")
          (lex "give")))
   (partic ((agent ((lex "John") (cat proper)))
            (affected ((lex "Mary") (cat proper)))
            (possessor {^ affected})))))


(def-test t73
  "To whom will John give a blue book?"
  ((cat clause)
   (mood wh)
   (tense future)
   (scope {^ partic possessor})
   (proc ((type composite)
          (relation-type possessive)
          (lex "give")))
   (partic ((agent ((lex "John") (cat proper)))
            (possessor ((animate yes)))
            (possessed ((lex "book")
                        (cat common)
                        (definite no)
                        (describer === "blue")))))))
;; Note that scope being DOUBLE role, e.g. here Af/Pr
;; since af and pr unify into affected-carrier.
;; can choose either one in your input.


(def-test t74
  "What does Deborah prefer?"
  ((cat clause)
   (mood wh)
```

```
   (scope {^ partic phenomenon})
   (proc ((type mental) (lex "prefer")))
   (partic ((processor ((cat proper) (lex "Deborah")))))))


(def-test t75
  "Who worships Baal?"
  ((cat clause)
   (mood wh)
   (scope {^ partic processor})
   (proc ((type mental) (lex "worship")))
   (partic ((processor ((animate yes)))
            (phenomenon ((cat proper) (lex "Baal")))))))


(def-test t76
  ;; Test reason role with a clause
  "Why does he do it?"
  ((cat clause)
   (mood wh)
   (scope {^ circum reason})
   (proc ((type material)
          (lex "do")))
   (partic ((agent ((cat personal-pronoun)
                    (animate yes)
                    (gender masculine)
                    (person third)
                    (number singular)))
            (affected ((cat personal-pronoun)
                       (gender neuter)
                       (person third)
                       (number singular)))))))


(def-test t77
  "From where does the SIG-display marker move to the right?"
  ((cat clause)
   (mood wh)
   (scope {^ circum from-loc})
   (proc ((type composite)
          (relation-type locative)
          (agentive no)
          (lex "move")))
   (partic ((affected ((cat common) (lex "sig-display marker")))
            (located {^ affected})
            (location ((cat pp) (prep === to)
                       (np ((lex "right")))))))))


;; ------------------------------------------------------------
;; LONG DISTANCE DEPENDENCY:
;; ------------------------------------------------------------
(def-test t77quad
  "Who do you think won the prize?"
  ((cat clause)
   (mood wh)
   (scope {^ partic phenomenon partic possessor})
   (proc ((type mental)
          (object-clause that)
          (lex "think")))
   (partic ((processor ((cat personal-pronoun)
                        (animate yes)
                        (person second)
                        (number singular)))
            ;; Win interpreted as "get": Ag/Pd+Pos
            (phenomenon ((cat clause)
                         (binder ((gap yes)))
```

```
                        (proc ((type composite)
                               (relation-type possessive)
                               (tense past)
                               (lex "win")))
                        (partic ((possessor ((animate yes)))
                                 (agent {^ possessor})
                                 (possessed ((cat common)
                                             (lex "prize")))))))))


;; ------------------------------------------------------------
;; Bound
;; ------------------------------------------------------------

(store-verbs '( ("win" "wins" "won" "winning" "won")) )
(def-test t77bis
  "You think that John won the prize."
  ((cat clause)
   (proc ((type mental)
          (object-clause that)
          (lex "think")))
   (partic ((processor ((cat personal-pronoun)
                        (animate yes)
                        (person second)
                        (number singular)))
            ;; Win interpreted as "get": Ag/Ca+Pos
            (phenomenon ((cat clause)
                         (proc ((type composite)
                                (relation-type possessive)
                                (tense past)
                                (lex "win")))
                         (partic ((possessor ((animate yes)
                                              (cat proper)
                                              (lex "John")))
                                  (agent {^ possessor})
                                  (possessed ((cat common)
                                              (lex "prize")))))))))))

(def-test t77bisp
  "You think that the prize was won by John."
  ((cat clause)
   (proc ((type mental)
          (object-clause that)
          (lex "think")))
   (partic ((processor ((cat personal-pronoun)
                        (animate yes)
                        (person second)
                        (number singular)))
            ;; Win interpreted as "get": Ag/Ca+Pos
            (phenomenon ((cat clause)
                         (proc ((type composite)
                                (relation-type possessive)
                                (tense past)
                                (voice passive)
                                (lex "win")))
                         (agentless no)
                         (partic ((possessor ((animate yes)
                                              (cat proper)
                                              (lex "John")))
                                  (agent {^ possessor})
                                  (possessed ((cat common)
                                              (lex "prize")))))))))))

(def-test t77ter
  "Do you think that John won the prize?"
  ((cat clause)
   (mood yes-no)
   (proc ((type mental)
```

```
          (object-clause that)
          (lex "think")))
   (partic ((processor ((cat personal-pronoun)
                        (animate yes)
                        (person second)
                        (number singular)))
            ;; Win interpreted as "get": Ag/Ca+Pos
            (phenomenon ((cat clause)
                         (proc ((type composite)
                                (relation-type possessive)
                                (tense past)
                                (lex "win")))
                         (partic ((possessor ((animate yes)
                                              (cat proper)
                                              (lex "John")))
                                  (agent {^ possessor})
                                  (possessed ((cat common)
                                              (lex "prize")))))))))


(def-test t140
  "The customer persuaded the programmer that there is a bug."
  ((cat clause)
   (tense past)
   (proc ((type lexical)
          (lex "persuade")
          (subcat ((1 {^3 lex-roles persuader})
                   (2 {^3 lex-roles persuadee})
                   (3 {^3 lex-roles soa})
                   (1 ((cat np)))
                   (2 ((cat np)))
                   (3 ((cat clause)
                       (mood bound)
                       (binder === that)))))))
   (lex-roles ((persuader === customer)
               (persuadee === programmer)
               (soa ((proc ((type existential)))
                     (partic ((located ((cat common)
                                        (definite no)
                                        (lex "bug")))))))))))



;; ------------------------------------------------------------
;; Relative mood
;; ------------------------------------------------------------

(def-test t78
  "The box which is expensive."
  ((cat common)
   (head === box)
   (qualifier ((cat clause)
               (scope {^ partic carrier})
               (proc ((type ascriptive)))
               (partic ((attribute === "expensive")))))))



(def-test t79
  "The box which is a classic."
  ((cat common)
   (head === box)
   (animate no)
   (qualifier ((cat clause)
               (proc ((type ascriptive)))
               (scope {^ partic carrier})
               (partic ((attribute ((cat common)
```

```
                                  (definite no)
                                  (lex "classic")))))))))


(def-test t80
  "The man who is your father."
  ((cat common)
   (head === man)
   (animate yes)
   (qualifier ((cat clause)
               (scope {^ partic identified})
               (proc ((type ascriptive) (mode equative)))
               (partic ((identifier ((cat common)
                                     (possessor ((cat personal-pronoun)
                                                 (person second)))
                                     (head === "father")))))))))


(def-test t81
  "The man that your father is."
  ((cat common)
   (head === man)
   (animate yes)
   (qualifier ((cat clause)
               (scope {^ partic identifier})
               (proc ((type ascriptive) (mode equative)))
               (partic ((identified ((cat common)
                                     (possessor ((cat personal-pronoun)
                                                 (person second)))
                                     (head === "father")))))))))



(def-test t82
  "The house where your mother is."
  ((cat common)
   (head === house)
   (qualifier ((cat clause)
               (scope {^ partic location})
               (proc ((type locative)))
               (partic ((located ((cat common)
                                  (possessor ((cat personal-pronoun)
                                              (person second)))
                                  (head === "mother")))))))))



(def-test t83
  "The man who is in your house."
  ((cat common)
   (head === man)
   (animate yes)
   (qualifier ((cat clause)
               (scope {^ partic located})
               (proc ((type locative)))
               (partic
                ((location ((cat pp)
                            (prep === "in")
                            (np ((cat common)
                                 (possessor ((cat personal-pronoun)
                                             (person second)))
                                 (head === "house")))))))))))
```

```
(def-test t84
  "The plate which covers the opening."
  ((cat common)
   (head === plate)
   (qualifier ((cat clause)
               (scope {^ partic located})
               (proc ((type locative) (mode equative) (lex "cover")))
               (partic ((location ((cat common)
                                   (head === "opening")))))))))



(def-test t85
  "The hole which the seal covers."
  ((cat common)
   (head === hole)
   (animate no)
   (qualifier ((cat clause)
               (scope {^ partic location})
               (proc ((type locative) (mode equative) (lex "cover")))
               (partic ((located ((cat common)
                                  (head === "seal")))))))))



(def-test t86
  "The time when the game starts."
  ((cat common)
   (head === time)
   (qualifier ((cat clause)
               (scope {^ partic time})
               (proc ((type temporal)
                      (lex "start")))
               (partic ((located ((cat common)
                                   (head === "game")))))))))



(def-test t87
  "The thing that happens then."
  ((cat common)
   (head === thing)
   (qualifier ((cat clause)
               (restrictive yes)
               (scope {^ partic located})
               (proc ((type temporal) (lex "happen")))
               (partic ((time ((cat adv) (lex "then")))))))))



(def-test t88
  "The position from which the SIG-display marker moves to the right."
  ((cat common)
   (head === position)
   (qualifier
    ((cat clause)
     (scope {^ circum from-loc})
     (proc ((type composite)
            (relation-type locative)
            (agentive no)
            (lex "move")))
     (partic ((affected ((cat common) (lex "sig-display marker")))
              (located {^ affected})
              (location ((cat pp)
                         (prep === to)
                         (np ((lex "right")))))))))))
```

```
(def-test t89
  "The way which your sister is."
  ((cat common)
   (head === way)
   (qualifier ((cat clause)
               (scope {^ partic attribute})
               (proc ((type ascriptive)))
               (partic ((carrier ((cat common)
                                  (possessor ((cat personal-pronoun)
                                              (person second)))
                                  (head === "sister")))))))))



(def-test t90
  "The person with whom your sister is."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (proc ((type accompaniment)))
               (scope {^ partic location})
               (partic ((located ((cat common)
                                  (possessor ((cat personal-pronoun)
                                              (person second)))
                                  (head === "sister")))))))))



(def-test t91
  "The person who has a PhD."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (scope {^ partic possessor})
               (proc ((type possessive)))
               (partic ((possessed ((cat common)
                                    (definite no)
                                    (head === "PhD")))))))))



(def-test t92
  "The box which she has."
  ((cat common)
   (head === box)
   (animate no)
   (qualifier ((cat clause)
               (proc ((type possessive)))
               (scope {^ partic possessed})
               (partic ((possessor ((cat personal-pronoun)
                                    (gender feminine)
                                    (number singular)
                                    (person third)))))))))



(def-test t93
  "The person who owns this book."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (proc ((type possessive) (mode equative)))
               (scope {^ partic possessor})
```

```
               (partic ((possessed ((cat common)
                                    (distance near)
                                    (head === "book")))))))))


(def-test t93bis
  "The person by whom this book is owned."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (proc ((type possessive) (mode equative)
                      (voice passive)))
               (scope {^ partic possessor})
               (partic ((possessed ((cat common)
                                    (distance near)
                                    (head === "book")))))))))


(def-test t94
  "The box that she owns."
  ((cat common)
   (head === box)
   (qualifier ((cat clause)
               (restrictive yes)
               (proc ((type possessive) (mode equative)))
               (scope {^ partic possessed})
               (partic ((possessor ((cat personal-pronoun)
                                    (gender feminine)
                                    (person third)
                                    (number singular)))))))))


(def-test t95
  "The person who gives a blue book to Mary."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (scope {^ partic agent})
               (proc ((type composite)
                      (relation-type possessive)
                      (dative-prep "to")
                      (lex "give")))
               (partic ((possessed ((lex "book")
                                    (cat common)
                                    (definite no)
                                    (describer === "blue")))
                        (affected ((lex "Mary") (cat proper)))
                        (possessor {^ affected})))))))


(def-test t96
  "The box which John will give to Mary."
  ((cat common)
   (head === box)
   (qualifier ((cat clause)
               (tense future)
               (scope {^ partic possessed})
               (proc ((type composite)
                      (relation-type possessive)
                      (dative-prep "to")
                      (lex "give")))
               (partic ((agent ((lex "John") (cat proper)))
                        (affected ((lex "Mary") (cat proper)))
                        (possessor {^ affected})))))))


(def-test t97
```

```
  "The person to whom John will give a blue book."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (tense future)
               (scope {^ partic possessor})
               (proc ((type composite)
                      (relation-type possessive)
                      (dative-prep "to")
                      (lex "give")))
               (partic ((agent ((lex "John") (cat proper)))
                        (possessed ((lex "book")
                                    (cat common)
                                    (definite no)
                                    (describer === "blue")))))))))

(def-test t97bis
  "The person by whom a blue book will be given to John."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (tense future)
               (scope {^ partic agent})
               (proc ((type composite)
                      (relation-type possessive)
                      (dative-prep "to")
                      (voice passive)
                      (lex "give")))
               (partic ((possessor ((lex "John") (cat proper)))
                        (possessed ((lex "book")
                                    (cat common)
                                    (definite no)
                                    (describer === "blue")))))))))


(def-test t98
  "The box which Deborah prefers."
  ((cat common)
   (head === box)
   (qualifier ((cat clause)
               (scope {^ partic phenomenon})
               (proc ((type mental) (lex "prefer")))
               (partic ((processor ((cat proper)
                                    (lex "Deborah")))))))))


(def-test t99
  "The person who worships Baal."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier ((cat clause)
               (scope {^ partic processor})
               (proc ((type mental) (lex "worship")))
               (partic ((phenomenon ((cat proper)
                                     (lex "Baal")))))))))


;; ----------------------------------------------------------
;; Long distance dependency for relatives
;; ----------------------------------------------------------

(def-test t99bis
```

```
  "The person who you think won the prize."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier
    ((cat clause)
     (proc ((type mental)
            (object-clause that)
            (lex "think")))
     (scope {^ partic phenomenon partic possessor})
     (partic ((processor ((cat personal-pronoun)
                          (animate yes)
                          (person second)
                          (number singular)))
              ;; Win interpreted as "get": Ag/Ca+Pos
              (phenomenon
               ((cat clause)
                (binder ((gap yes)))
                (proc ((type composite)
                       (relation-type possessive)
                       (tense past)
                       (lex "win")))
                (partic ((agent {^ possessor})
                         (possessed ((cat common)
                                     (lex "prize")))))))))))))

(def-test t99ter
  "The person by whom you think the prize was won."
  ((cat common)
   (head === person)
   (animate yes)
   (qualifier
    ((cat clause)
     (proc ((type mental)
            (object-clause that)
            (lex "think")))
     (scope {^ partic phenomenon partic possessor})
     (partic
      ((processor ((cat personal-pronoun)
                   (animate yes)
                   (person second)
                   (number singular)))
       ;; Win interpreted as "get": Ag/Ca+Pos
       (phenomenon ((cat clause)
                    (binder ((gap yes)))
                    (proc ((type composite)
                           (voice passive)
                           (relation-type possessive)
                           (tense past)
                           (lex "win")))
                    (partic ((agent {^ possessor})
                             (possessed ((cat common)
                                         (lex "prize")))))))))))))


;; ============================================================
;; Non-Finite mood
;; ============================================================

(def-test t26
  "Take the hammer and hit the nail."
  ((cat clause)
   (complex conjunction)
   (common ((mood imperative)))
   (distinct
    ~(((proc ((type material) (lex "take")))
       (partic ((affected ((cat common) (head === "hammer"))))))
      ((proc ((type material) (lex "hit")))
```

```
       (partic ((affected ((cat common) (head === "nail")))))))))

(def-test t28
  ;; t28 fails because epistemic-modality is not compatible
  ;; with imperative.
  "<fail>"
  ((cat clause)
   (complex conjunction)
   (common ((mood imperative)))
   (distinct
    ~(((proc ((type material) (lex "take")))
       (epistemic-modality possible)
       (partic ((affected ((cat common) (head === "hammer"))))))
      ((proc ((type material) (lex "hit")))
       (partic ((affected ((cat common) (head === "nail")))))))))



;; Present-participle and infinitive
;; (both governed by the verb "cause")
(def-test t13
  "the power level's increasing causes the sig-display marker to
   move to the right."
  ((cat clause)
   (proc ((type lexical)
          (lex "cause")
          ;; These two features are a short notation for the
          ;; syntactic constraints described in the subcat frame below
          (subject-clause present-participle)
          (object-clause infinitive)
          (subcat ((1 {^3 lex-roles influence})
                   (2 {^3 lex-roles influenced})
                   (3 {^3 lex-roles soa})
                   (1 ((alt (((cat np))
                             ((cat clause)
                              (mood present-participle))))))
                   (2 ((cat np)))
                   (3 ((cat clause)
                       (mood infinitive)
                       (oblique ((1 ((index {^4 2 index}) ; controlled
                                     (gap yes)))))))))))
   (lex-roles
    ((influence ((cat clause)
                 (proc ((type material)
                        (lex "increase")
                        (agentive no)))
                 (partic ((affected ((cat common)
                                     (lex "power level")))))))
     (influenced ((cat common) (index ((concept sdmarker)))
                  (classifier === SIG-display) (lex "marker")))
     (soa ((cat clause)
           (proc ((type composite) (relation-type locative)
                  (agentive no) (lex "move")))
           (partic
            ((affected ((cat common) (index ((concept sdmarker)))))
             (located {^ affected})
             (location ((cat pp)
                        (prep === "to")
                        (np ((cat common) (lex "right")))))))))))



;; Subjunctive
(def-test t219
  "Karen demanded that SURGE be used in 1992."
  ((cat clause)
   (process
    ((type lexical)
```

```
  (lex "demand")
  (tense past)
  (subcat ((1 {^3 lex-roles influence})
           (2 {^3 lex-roles soa})
           (2 ((cat clause) (mood bound-subjunctive)))))))
(lex-roles
 ((influence ((cat proper) (lex "Karen")))
  (soa ((process ((type material) (tense past) (lex "use")))
        (partic ((affected ((lex "SURGE") (cat proper)))))))))
(circum ((time ((cat proper) (lex "1992") (time-type "in")))))))
```

A.5. Modality and Polarity

Modality can be expressed in semantic terms (using labels such as possibility or inference) or by directly specifying a modal. Polarity is specified by adding the feature polarity at either the top level of the clause or in the verb. Negation has currently no scope in SURGE.

The SIG-display marker can move to the right.
For her to do it must be a bold statement.
To be innocent, you must do it for him to eat.
The man whom I may know.
Who does not have a PhD?
Has not John given it to Mary?
John did give Mary a blue book.

```
;; ============================================================
;; Modality
;; ============================================================

;; Possible mapped to can
(def-test t10bis
  "The SIG-display marker can move to the right."
  ((cat clause)
   (epistemic-modality possible)
   (proc ((type composite)
          (relation-type locative)
          (agentive no)
          (lex "move")))
   (partic ((affected ((cat common)
                       (lex "SIG-display marker")))
            (located {^ affected})
            (location ((cat pp)
                       (prep === "to")
                       (np ((cat common) (lex "right")))))))))


;; Inference mapped to must
(def-test t15-bis
  "For her to do it must be a bold statement."
  ((cat clause)
   (epistemic-modality inference)
   (proc ((type ascriptive)
          (mode attributive)))
   (partic ((carrier ((cat clause)
                      (proc ((lex "do")
                             (type material)))
                      (partic ((agent ((cat personal-pronoun)
                                       (gender feminine)))
                               (affected ((cat personal-pronoun)
                                          (gender neuter)))))))
            (attribute ((cat common)
                        (definite no)
                        (lex "statement")
                        (describer === "bold")))))))


;; Must as duty
(def-test t51
  "To be innocent, you must do it for him to eat."
  ((cat clause)
   (deontic-modality duty)
   (process ((lex "do")
             (type material)))
   (partic ((agent ((cat personal-pronoun)
                    (animate yes)
                    (person second)
                    (number singular)))
            (affected ((cat personal-pronoun)
                       (gender neuter)
                       (person third)
                       (number singular)))))
   (circum
    ((purpose ((cat clause)
               (keep-for no)
               (keep-in-order no)
               (process ((type ascriptive)
                         (relation-type attributive)))
               (partic
                ((carrier ((cat personal-pronoun)
                           (semantics {partic agent semantics})))
                 (attribute === innocent)))))
     (behalf ((cat clause)
              (process ((type material)
```

```
                    (lex "eat")))
               (partic ((agent ((cat personal-pronoun)
                                (animate yes)
                                (person third)
                                (gender masculine)
                                (number singular)))))))))))


;; Can also directly specify the modal (may)
(def-test t16bis
  "The man whom I may know."
  ((cat common)
   (head === "man")
   (animate yes)
   (qualifier ((cat clause)
               (epistemic-modality "may")
               (restrictive no)
               ;; (scope ((role phenomenon)))
               (scope {^ participants phenomenon})
               (proc ((type mental)
                      (lex "know")
                      (transitive yes)))
               (participants
                ((processor ((cat personal-pronoun)
                             (person first)))))))))


;; Interaction modality/mood
(def-test t28
  ;; t28 fails because epistemic-modality is not
  ;; compatible with imperative.
  "<fail>"
  ((cat clause)
   (complex conjunction)
   (common ((mood imperative)))
   (distinct
    ~(((proc ((type material) (lex "take")))
       (epistemic-modality possible)
       (partic ((affected ((cat common) (head === "hammer"))))))
      ((proc ((type material) (lex "hit")))
       (partic ((affected ((cat common) (head === "nail")))))))))))


;; ============================================================
;; Polarity
;; ============================================================

;; Polarity indicated by the polarity feature
(def-test t67n
  "Who does not have a PhD?"
  ((cat clause)
   (mood wh)
   (polarity negative)
   (scope {^ partic possessor})
   (proc ((type possessive)))
   (partic ((possessor ((animate yes)))
            (possessed ((cat common) (definite no)
                        (head === "PhD")))))))


;; Have as an auxiliary
(def-test t2bis
  "Has not John given it to Mary?"
  ((cat clause)
   (tense present-perfect)
   (polarity negative)
   (proc ((type composite)
```

```
          (relation-type possessive)
          (lex "give")))
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessor {^ affected})
            (possessed ((cat pronoun)))))))


;; Insistence
(def-test t54
  "John did give Mary a blue book."
  ((cat clause)
   (insistence yes)
   (proc ((type composite)
          (relation-type possessive)
          (dative-prep "to")
          (lex "give")))
   (tense past)
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessor {^ affected})
            (possessed ((lex "book")
                        (cat common)
                        (definite no)
                        (describer === "blue")))))))
```

A.6. Circumstantials

Circumstantials are the complements in the clause which are not inherently bound to the process type. They are most often realized by adjuncts at the syntactic level. The following circumstantials are recognized by SURGE:

- Instrument
- Accompaniment
- Manner
- Purpose
- Reason
- Behalf
- Time
- Temporal-background
- At-location
- To-location
- From-location
- In-location
- On-location

For each circumstantial, SURGE determines what syntactic complement can be used to realize it, its position in the clause, the category of the syntactic complement, and defaults for prepositions and subordinators. The following input examples illustrate most of these tasks:

In order to install the battery, clean the box.
To be innocent, you must do it for him to eat.
Hit the nail with the hammer.

Hit the nail using the hammer.
Take the nail off using a screwdriver.
The nail is taken off using a screwdriver.
Take the nail off using a screwdriver.
He does it because of you.
He does it because he likes you.
He does it when he likes you.
When hitting you, Sam hurt Mary.
When properly done, it can be safe.
He scored 39 points for the Lakers.
This DLC refinement activated ALL-DLC for CSA 2119 in 1992Q1.

```
;; Different variations of the purpose circumstantial
;; Default is ‘‘in order to’’ in front of clause.
(def-test t29
  "In order to install the battery, clean the box."
  ((cat clause)
   (mood imperative)
   (proc ((type material)
          (lex "clean")))
   (partic ((affected ((cat common) (head === "box")))
            (agent ((semantics ((index ((concept c-user)))))))))
   (circum ((purpose ((proc ((type material)
                             (lex "install")))
                      (partic ((affected ((cat common)
                                          (head === "battery")))
                               (agent {partic agent}))))))))))


(def-test t51
  ;; Reason role vs. Purpose: the two are distincts and can cooccur.
  "To be innocent, you must do it for him to eat."
  ((cat clause)
   (deontic-modality duty)
   (process ((lex "do")
             (type material)))
   (partic ((agent ((cat personal-pronoun) (animate yes)
                    (person second) (number singular)))
            (affected ((cat personal-pronoun) (gender neuter)
                       (person third) (number singular)))))
   (circum
    ((purpose ((cat clause)
               (keep-for no)
               (keep-in-order no)
               (process ((type ascriptive)
                         (relation-type attributive)))
               (partic
                ((carrier ((cat personal-pronoun)
                           (index {partic agent index})))
                 (attribute === innocent)))))
     (behalf ((cat clause)
              (process ((type material) (lex "eat")))
              (partic ((agent ((cat personal-pronoun) (animate yes)
                               (person third) (gender masculine)
                               (number singular)))))))))))


;; Different ways of specifying the preposition
;; to use for a circumstantial mapped to a PP:
;; Default
(def-test t33
  "Hit the nail with the hammer."
  ((cat clause)
   (mood imperative)
   (proc ((type material) (lex "hit")))
   (partic ((affected ((cat common) (head === "nail")))))
   (circum ((instrument ((cat common) (head === "hammer")))))))


;; If it depends on the verb, specify it in the verb.
(def-test t34
  "Hit the nail using the hammer."
  ((cat clause)
   (mood imperative)
   (proc ((type material) (lex "hit")
          (instrument-prep ((lex "using")))))
   (partic ((affected ((cat common) (head === nail)))))
   (circum ((instrument ((cat common) (head === "hammer")))))))
```

```
;; Note the use of particle for "take off"
(def-test t35
  "Take the nail off using a screwdriver."
  ((cat clause)
   (mood imperative)
   (proc ((type material) (lex "take") (particle "off")
          (instrument-prep ((lex "using")))))
   (partic ((affected ((cat common) (head === nail)))))
   (circum ((instrument ((cat common) (head === screwdriver)
                         (definite no)))))))



(def-test t36
  "The nail is taken off using a screwdriver."
  ((cat clause)
   (proc ((type material) (voice passive)
          (lex "take") (particle "off")
          (instrument-prep ((lex "using")))))
   (partic ((affected ((cat common) (head === nail)))))
   (circum ((instrument ((cat common)
                         (head === screwdriver)
                         (definite no)))))))



;; another way of specifying the prep for the role instrument.
;; If it depends on the object, put it in the object.
(def-test t37
  "Take the nail off using a screwdriver."
  ((cat clause)
   (mood imperative)
   (proc ((type material) (lex "take") (particle "off")))
   (partic ((affected ((cat common) (head === nail)))))
   (circum ((instrument ((cat common)
                         (prep ((lex "using")))
                         (head === screwdriver) (definite no)))))))


(def-test t39
  ;; Test reason role
  "He does it because of you."
  ((cat clause)
   (process ((type material) (lex "do")))
   (partic ((agent ((cat personal-pronoun)
                    (animate yes) (gender masculine)
                    (person third) (number singular)))
            (affected ((cat personal-pronoun) (gender neuter)
                       (person third) (number singular)))))
   (circum ((reason ((cat personal-pronoun) (gender masculine)
                     (person second) (number singular)))))))



(def-test t40
  ;; Test reason role with a clause
  "He does it because he likes you."
  ((cat clause)
   (process ((lex "do")
             (type material)))
   (partic ((agent ((cat personal-pronoun) (animate yes)
                    (gender masculine) (person third)
                    (number singular)))
            (affected ((cat personal-pronoun) (gender neuter)
                       (person third) (number singular)))))
   (circum ((reason ((cat clause)
                     (process ((type mental) (lex "like")))
                     (partic ((processor {partic agent})
```

```
                              (phenomenon ((cat personal-pronoun)
                                           (person second)
                                           (number singular)))))))))))


;; Test temporal-background role: three forms of clause
;; declarative, present-participle and past-participle.
;; Each has different constraints on which element of the matrix clause
;; is coindexed with the subject of the circumstantial.
(def-test t40bis
  "He does it when he likes you."
  ((cat clause)
   (process ((lex "do") (type material)))
   (partic ((agent ((cat personal-pronoun) (animate yes)
                    (gender masculine) (person third)
                    (number singular)))
            (affected ((cat personal-pronoun) (gender neuter)
                       (person third) (number singular)))))
   (circum ((temporal-background
             ((cat clause)
              (mood declarative)
              (process ((type mental) (lex "like")))
              (partic ((processor {partic agent})
                       (phenomenon ((cat personal-pronoun)
                                    (person second)
                                    (number singular)))))))))))


(store-verbs '( ("hurt" "hurts" "hurt" "hurting" "hurt")) )
(def-test t40ter
  ;; Test temporal-background role
  "When hitting you, Sam hurt Mary."
  ((cat clause)
   (process ((lex "hurt") (tense past) (type material)))
   (partic ((agent ((cat proper) (animate yes) (index sam1)
                    (gender masculine) (lex "Sam") (number singular)))
            (affected ((cat proper) (lex "Mary") (number singular)))))
   (circum
    ((temporal-background
      ((cat clause)
       (position front)
       (mood present-participle)
       (process ((type material) (lex "hit")))
       (partic ((agent ((index sam1)))
                (affected ((cat personal-pronoun) (person second)
                           (number singular)))))))))))


(def-test t40quad
  ;; Test temporal-background role
  "When properly done, it can be safe."
  ((cat clause)
   (process ((type ascriptive)))
   (epistemic-modality possible)
   (partic ((carrier ((cat personal-pronoun) (gender neuter)
                      (person third)
                      (index id1) ; unique identifier for this referent
                      (number singular)))
            (attribute ((lex "safe") (cat adj)))))
   (circum ((temporal-background
             ((cat clause)
              (position front)
              (mood past-participle)
              (process ((type material) (lex "do")))
              (partic ((affected ((index {^5 partic carrier index})))))
              (circum ((manner ((lex "properly")))))))))))
```

```
(def-test t41
  ;; Test reason role
  "He scored 39 points for the Lakers."
  ((cat clause)
   (tense past)
   (process ((type material) (lex "score")))
   (partic ((agent ((cat personal-pronoun) (animate yes)
                    (gender masculine) (person third)
                    (number singular)))
            (affected ((cat common) (definite no)
                       (cardinal ((value 39) (digit yes)))
                       (lex "point")))))
   (circum ((behalf ((cat proper) (number plural) (lex "Laker")))))))


(def-test t217
  "This DLC refinement activated ALL-DLC for CSA 2119 in 1992Q1."
  ((cat clause)
   (process ((type material) (tense past) (lex "activate")))
   (partic ((agent ((distance near) (lex "refinement")
                    (classifier ((lex "DLC")))))
            (affected ((cat proper) (lex "ALL-DLC")))))
   (circum ((behalf ((cat proper) (lex "CSA 2119")))
            (time ((cat proper) (lex "1992Q1") (time-type "in")))))))
```

A.7. Tense

SURGE supports the 36 tenses identified in [Halliday 85]. The following list shows the same input generated with all 36 tenses and for each tense, with both active and passive voice, positive and negative polarity, and declarative and yes-no interrogative mood. A simple function iterates over these 36x7 forms for the same FD.

```
(def-test t2bis
  "John gives it to Mary."
  ((cat clause)
   (proc ((type composite)
          (relation-type possessive)
          (lex "give")))
   (partic ((agent ((cat proper) (lex "John")))
            (affected ((cat proper) (lex "Mary")))
            (possessor {^ affected})
            (possessed ((cat pronoun)))))))


=======================
TENSE-1
John gave it to Mary.
It was given to Mary.
John did not give it to Mary.
Did John give it to Mary?
It was not given to Mary.
Did not John give it to Mary?
Was not it given to Mary?

=======================
TENSE-2
John gives it to Mary.
It is given to Mary.
John does not give it to Mary.
Does John give it to Mary?
It is not given to Mary.
Does not John give it to Mary?
Is not it given to Mary?

=======================
TENSE-3
John will give it to Mary.
It will be given to Mary.
John will not give it to Mary.
Will John give it to Mary?
It will not be given to Mary.
Will not John give it to Mary?
Will not it be given to Mary?

=======================
TENSE-4
John had given it to Mary.
It had been given to Mary.
John had not given it to Mary.
Had John given it to Mary?
It had not been given to Mary.
Had not John given it to Mary?
Had not it been given to Mary?

=======================
TENSE-5
John has given it to Mary.
It has been given to Mary.
John has not given it to Mary.
Has John given it to Mary?
It has not been given to Mary.
Has not John given it to Mary?
Has not it been given to Mary?

=======================
TENSE-6
John will have given it to Mary.
It will have been given to Mary.
John will not have given it to Mary.
Will John have given it to Mary?
```

It will not have been given to Mary.
Will not John have given it to Mary?
Will not it have been given to Mary?

==========
TENSE-7
John was giving it to Mary.
It was being given to Mary.
John was not giving it to Mary.
Was John giving it to Mary?
It was not being given to Mary.
Was not John giving it to Mary?
Was not it being given to Mary?

==========
TENSE-8
John is giving it to Mary.
It is being given to Mary.
John is not giving it to Mary.
Is John giving it to Mary?
It is not being given to Mary.
Is not John giving it to Mary?
Is not it being given to Mary?

==========
TENSE-9
John will be giving it to Mary.
It will be being given to Mary.
John will not be giving it to Mary.
Will John be giving it to Mary?
It will not be being given to Mary.
Will not John be giving it to Mary?
Will not it be being given to Mary?

==========
TENSE-10
John was going to give it to Mary.
It was going to be given to Mary.
John was not going to give it to Mary.
Was John going to give it to Mary?
It was not going to be given to Mary.
Was not John going to give it to Mary?
Was not it going to be given to Mary?

==========
TENSE-11
John is going to give it to Mary.
It is going to be given to Mary.
John is not going to give it to Mary.
Is John going to give it to Mary?
It is not going to be given to Mary.
Is not John going to give it to Mary?
Is not it going to be given to Mary?

==========
TENSE-12
John will be going to give it to Mary.
It will be going to be given to Mary.
John will not be going to give it to Mary.
Will John be going to give it to Mary?
It will not be going to be given to Mary.
Will not John be going to give it to Mary?
Will not it be going to be given to Mary?

==========
TENSE-13
John was going to have given it to Mary.
It was going to have been given to Mary.

John was not going to have given it to Mary.
Was John going to have given it to Mary?
It was not going to have been given to Mary.
Was not John going to have given it to Mary?
Was not it going to have been given to Mary?

==========
TENSE-14
John is going to have given it to Mary.
It is going to have been given to Mary.
John is not going to have given it to Mary.
Is John going to have given it to Mary?
It is not going to have been given to Mary.
Is not John going to have given it to Mary?
Is not it going to have been given to Mary?

==========
TENSE-15
John will be going to have given it to Mary.
It will be going to have been given to Mary.
John will not be going to have given it to Mary.
Will John be going to have given it to Mary?
It will not be going to have been given to Mary.
Will not John be going to have given it to Mary?
Will not it be going to have been given to Mary?

==========
TENSE-16
John had been giving it to Mary.
It had been being given to Mary.
John had not been giving it to Mary.
Had John been giving it to Mary?
It had not been being given to Mary.
Had not John been giving it to Mary?
Had not it been being given to Mary?

==========
TENSE-17
John has been giving it to Mary.
It has been being given to Mary.
John has not been giving it to Mary.
Has John been giving it to Mary?
It has not been being given to Mary.
Has not John been giving it to Mary?
Has not it been being given to Mary?

==========
TENSE-18
John will have been giving it to Mary.
It will have been being given to Mary.
John will not have been giving it to Mary.
Will John have been giving it to Mary?
It will not have been being given to Mary.
Will not John have been giving it to Mary?
Will not it have been being given to Mary?

==========
TENSE-19
John was going to be giving it to Mary.
It was going to be being given to Mary.
John was not going to be giving it to Mary.
Was John going to be giving it to Mary?
It was not going to be being given to Mary.
Was not John going to be giving it to Mary?
Was not it going to be being given to Mary?

==========
TENSE-20

John is going to be giving it to Mary.
It is going to be being given to Mary.
John is not going to be giving it to Mary.
Is John going to be giving it to Mary?
It is not going to be being given to Mary.
Is not John going to be giving it to Mary?
Is not it going to be being given to Mary?

==========
TENSE-21
John will be going to be giving it to Mary.
It will be going to be being given to Mary.
John will not be going to be giving it to Mary.
Will John be going to be giving it to Mary?
It will not be going to be being given to Mary.
Will not John be going to be giving it to Mary?
Will not it be going to be being given to Mary?

==========
TENSE-22
John had been going to give it to Mary.
It had been going to be given to Mary.
John had not been going to give it to Mary.
Had John been going to give it to Mary?
It had not been going to be given to Mary.
Had not John been going to give it to Mary?
Had not it been going to be given to Mary?

==========
TENSE-23
John has been going to give it to Mary.
It has been going to be given to Mary.
John has not been going to give it to Mary.
Has John been going to give it to Mary?
It has not been going to be given to Mary.
Has not John been going to give it to Mary?
Has not it been going to be given to Mary?

==========
TENSE-24
John will have been going to give it to Mary.
It will have been going to be given to Mary.
John will not have been going to give it to Mary.
Will John have been going to give it to Mary?
It will not have been going to be given to Mary.
Will not John have been going to give it to Mary?
Will not it have been going to be given to Mary?

==========
TENSE-25
John had been going to have given it to Mary.
It had been going to have been given to Mary.
John had not been going to have given it to Mary.
Had John been going to have given it to Mary?
It had not been going to have been given to Mary.
Had not John been going to have given it to Mary?
Had not it been going to have been given to Mary?

==========
TENSE-26
John has been going to have given it to Mary.
It has been going to have been given to Mary.
John has not been going to have given it to Mary.
Has John been going to have given it to Mary?
It has not been going to have been given to Mary.
Has not John been going to have given it to Mary?
Has not it been going to have been given to Mary?

==========

TENSE-27
John will have been going to have given it to Mary.
It will have been going to have been given to Mary.
John will not have been going to have given it to Mary.
Will John have been going to have given it to Mary?
It will not have been going to have been given to Mary.
Will not John have been going to have given it to Mary?
Will not it have been going to have been given to Mary?

==========
TENSE-28
John was going to have been giving it to Mary.
It was going to have been being given to Mary.
John was not going to have been giving it to Mary.
Was John going to have been giving it to Mary?
It was not going to have been being given to Mary.
Was not John going to have been giving it to Mary?
Was not it going to have been being given to Mary?

==========
TENSE-29
John is going to have been giving it to Mary.
It is going to have been being given to Mary.
John is not going to have been giving it to Mary.
Is John going to have been giving it to Mary?
It is not going to have been being given to Mary.
Is not John going to have been giving it to Mary?
Is not it going to have been being given to Mary?

==========
TENSE-30
John will be going to have been giving it to Mary.
It will be going to have been being given to Mary.
John will not be going to have been giving it to Mary.
Will John be going to have been giving it to Mary?
It will not be going to have been being given to Mary.
Will not John be going to have been giving it to Mary?
Will not it be going to have been being given to Mary?

==========
TENSE-31
John had been going to be giving it to Mary.
It had been going to be being given to Mary.
John had not been going to be giving it to Mary.
Had John been going to be giving it to Mary?
It had not been going to be being given to Mary.
Had not John been going to be giving it to Mary?
Had not it been going to be being given to Mary?

==========
TENSE-32
John has been going to be giving it to Mary.
It has been going to be being given to Mary.
John has not been going to be giving it to Mary.
Has John been going to be giving it to Mary?
It has not been going to be being given to Mary.
Has not John been going to be giving it to Mary?
Has not it been going to be being given to Mary?

==========
TENSE-33
John will have been going to be giving it to Mary.
It will have been going to be being given to Mary.
John will not have been going to be giving it to Mary.
Will John have been going to be giving it to Mary?
It will not have been going to be being given to Mary.
Will not John have been going to be giving it to Mary?
Will not it have been going to be being given to Mary?

```
=======================
TENSE-34
John had been going to have been giving it to Mary.
It had been going to have been being given to Mary.
John had not been going to have been giving it to Mary.
Had John been going to have been giving it to Mary?
It had not been going to have been being given to Mary.
Had not John been going to have been giving it to Mary?
Had not it been going to have been being given to Mary?

=======================
TENSE-35
John has been going to have been giving it to Mary.
It has been going to have been being given to Mary.
John has not been going to have been giving it to Mary.
Has John been going to have been giving it to Mary?
It has not been going to have been being given to Mary.
Has not John been going to have been giving it to Mary?
Has not it been going to have been being given to Mary?

=======================
TENSE-36
John will have been going to have been giving it to Mary.
It will have been going to have been being given to Mary.
John will not have been going to have been giving it to Mary.
Will John have been going to have been giving it to Mary?
It will not have been going to have been being given to Mary.
Will not John have been going to have been giving it to Mary?
Will not it have been going to have been being given to Mary?
FUG5>
```

A.8. Conjunction and Ellipsis

Conjunction in SURGE is uniformly handled. All syntactic categories can be enriched by a feature (complex conjunction), which indicate that a complex constituent is to be generated. In this case, the input for the constituent must contain two main features: common and distinct. Common contains the FD which is common to all conjuncts. Distinct is an ordered list of all the conjuncts. A conjunction can also be specified. By default, *and* is selected.

In the case of the clause, SURGE performs ellipsis when all conjuncts share a role or when the same verb is used in all conjuncts. The following examples illustrate this aspect of the grammar:

Take the hammer and hit the nail.
First, take the hammer and then, hit the nail.
John took the hammer and hit the nail.
John took the hammer and Mary the nail.
This DLC refinement activates CSA 4703 in 1992Q1 and CSA 4704 in 1992Q2.

```
(def-test t26
  "Take the hammer and hit the nail."
  ((cat clause)
   (complex conjunction)
   (common ((mood imperative)))
   (distinct
    ~(((proc ((type material) (lex "take")))
       (partic ((affected ((cat common) (head === "hammer"))))))
      ((proc ((type material) (lex "hit")))
       (partic ((affected ((cat common) (head === "nail")))))))))))


(def-test t27
  "First, take the hammer and then, hit the nail."
  ((cat clause)
   (complex conjunction)
   (common ((mood imperative)))
   (distinct
    ~(((proc ((type material) (lex "take")))
       (time-relater === "first")
       (partic ((affected ((cat common) (head === "hammer"))))))
      ((proc ((type material) (lex "hit")))
       (time-relater === "then")
       (partic ((affected ((cat common) (head === "nail")))))))))))


;; Subject ellipsis
(def-test t28
  "John took the hammer and hit the nail."
  ((cat clause)
   (complex conjunction)
   (common ((tense past)))
   (distinct
    ~(((proc ((type material) (lex "take")))
       (partic ((agent ((cat proper) (lex "John")
                        (index ((concept j1)))))
                (affected ((cat common) (head === "hammer"))))))
      ((proc ((type material) (lex "hit")))
       (partic ((agent ((cat proper) (lex "John")
                        (index ((concept j1)))))
                (affected ((cat common) (head === "nail")))))))))))


;; Verb ellipsis
(def-test t29
  "John took the hammer and Mary the nail."
  ((cat clause)
   (complex conjunction)
   (common ((tense past)))
   (distinct
    ~(((proc ((type material) (lex "take")))
       (partic ((agent ((cat proper) (lex "John")
                        (index ((concept j1)))))
                (affected ((cat common) (head === "hammer"))))))
      ((proc ((type material) (lex "take")))
       (partic ((agent ((cat proper) (lex "Mary")
                        (index ((concept m1)))))
                (affected ((cat common) (head === "nail")))))))))))



;; Verb AND subject ellipsis.
(def-test t213
  "This DLC refinement activates CSA 4703 in 1992Q1
   and CSA 4704 in 1992Q2."
  ((cat clause)
   (complex conjunction)
   (distinct
```

```
~(
  ((process ((type material)
             (lex "activate")))
   (partic ((agent ((cat common)
                    (index ((concept DLC-R)))
                    (classifier ((lex "DLC")))
                    (lex "refinement")
                    (animate no) (distance near)
                    (number singular)))
            (affected ((cat proper) (index ((concept CSA4703)))
                       (lex "CSA 4703")))))
   (circum ((time ((cat proper) (time-type "in")
                   (head === "1992Q1"))))))
  ((process ((type material)
             (lex "activate")))
   (partic ((agent ((cat common)
                    (index ((concept DLC-R)))
                    (classifier ((lex "DLC")))
                    (lex "refinement")
                    (animate no) (distance near)
                    (number singular)))
            (affected ((cat proper) (index ((concept CSA4704)))
                       (lex "CSA 4704")))))
   (circum ((time ((cat proper) (time-type "in")
                   (head === "1992Q2")))))))))
```

Appendix B
SURGE Coverage: the Determiner Sequence

This appendix demonstrates the coverage of the part of the SURGE grammar dealing with the determiner sequence. This part of the grammar is described in Sect.5.4. The part of the lexical chooser dealing with determiners is described in Sect.6.6. The determiner sequence has the specificity that it is a closed-system: output of the lexical chooser does not consist in lexical items but in a set of features which SURGE uses to select one of a closed set of determiners.

The features used in SURGE to control the generation of the determiner sequence are the following (cf. Sect.5.4.3 p.144):

- **Definite**: yes, no.
- **Countable**: yes, no.
- **Partitive**: yes, no.
- **Interrogative**: yes, no.
- **Possessive**: yes, no.
- **Number**: singular, dual, plural.[65]
- **Reference-number**: singular, dual, plural.
- **Distance**: far, near.
- **Selective**: yes, no.
- **Total**: +, -, no.
- **Exact**: yes, no.
- **Orientation**: +, -, none.
- **Degree**: +, -, none.
- **Evaluative**: yes, no.
- **Evaluation**: +, -.
- **Status**: none, same, different, mentioned, specific, usual, entire or a lexicalized item.
- **Case**: objective, subjective, genitive.
- **Head-cat**: pronoun, common, proper.
- **denotation-class**: quantity, season, institution, transportation, meal, illness.
- **Comparative**: yes, no.
- **Superlative**: yes, no.
- **Cardinal**: number.

[65]Dual is necessary to account for the determiners *both*, *either* and *neither*.

- **Ordinal**: number or last, next (+), previous (-).
- **Fraction**: numerator, denominator.

The use of these features is demonstrated in the following examples, which illustrate the coverage of SURGE. Examples include cases of:

- Quantifier determiners
- Multipliers and fractions
- Zero articles for certain semantic classes
- Interaction of determiners and pronouns
- Interaction of determiners with proper nouns
- Argumentative determiners
- Use of status (deictic2) adjective in the determiner sequence

```
;; ============================================================
;; Test of determiner sequence
;; ============================================================

(def-test t160
  "The car."
  ((cat np)
   (definite yes)
   (lex "car")))

(def-test t161
  "The cars."
  ((cat np)
   (definite yes)
   (number plural)
   (lex "car")))

(def-test t162
  "A car."
  ((cat np)
   (definite no)
   (lex "car")))

(def-test t163
  "Cars."
  ((cat np)
   (definite no)
   (number plural)
   (lex "car")))

(def-test t164
  "Which car."
  ((cat np)
   (interrogative yes)
   (lex "car")))

(def-test t165
  "What car."
  ((cat np)
   (interrogative yes)
   (selective no)
   (lex "car")))

(def-test t166
  "Whose car."
  ((cat np)
   (interrogative yes)
   (possessive yes)
   (lex "car")))

(def-test t167
  "My car."
  ((cat np)
   (lex "car")
   (possessor ((cat personal-pronoun)
               (person first)))))

(def-test t168
  "John's car."
  ((cat np)
   (lex "car")
   (possessor ((cat proper)
               (lex "John")))))

(def-test t169
  "The man's cars."
  ((cat np)
   (number plural)
```

```
   (lex "car")
   (possessor ((cat np)
               (definite yes)
               (lex "man")))))

(def-test t170
  "Which man's car."
  ((cat np)
   (interrogative yes)
   (lex "car")
   (possessor ((cat np)
               (lex "man")))))


(def-test t171
  "This car."
  ((cat common)
   (distance near)
   (lex "car")))

(def-test t172
  "That car."
  ((cat common)
   (distance far)
   (lex "car")))


;; -------------------------------------------------------------
;; quantifier det
;; -------------------------------------------------------------

(def-test t173
  "All cars."
  ((cat common)
   (number plural)
   (definite no)
   (total +)
   (lex "car")))

(def-test t174
  "No car."
  ((cat common)
   (number singular)
   (total -)
   (lex "car")))

(def-test t175
  "No cars."
  ((cat common)
   (number plural)
   (total -)
   (lex "car")))

(def-test t176
  ("Each car."
   "Every car.")
  ((cat common)
   (lex "car")
   (number singular)
   (total +)))

(def-test t177
  "Both cars."
  ((cat common)
   (lex "car")
   (definite no)
   (number dual)
   (total +)))
```

```
(def-test t178
  "Neither car."
  ((cat common)
   (lex "car")
   (reference-number dual)
   (total -)))

(def-test t179
  "One car."
  ((cat common)
   (lex "car")
   (total none)
   (selective yes)
   (number singular)))

(def-test t180
  "Either car."
  ((cat common)
   (lex "car")
   (total none)
   (reference-number dual)
   (selective yes)))

(def-test t181
  "Some cars."
  ((cat common)
   (lex "car")
   (number plural)
   (total none)
   (selective yes)))



;; ------------------------------------------------------------
;; Multipliers and fractions
;; ------------------------------------------------------------

(def-test t182
  "Two-thirds of the cars."
  ((cat common)
   (lex "car")
   (fraction ((num 2)
              (den 3)))
   (definite yes)
   (number plural)))

(def-test t183
  "2/3 of the car."
  ((cat common)
   (lex "car")
   (fraction ((num 2) (den 3) (digit yes)))
   (definite yes)))

(def-test t184
  "Three times the butter."
  ((cat common)
   (lex "butter")
   (countable no)
   (multiplier ((value 3)))
   (definite yes)))

(def-test t185
  ("Twice his salary."
   "Double his salary.")
  ((cat common)
   (lex "salary")
   (denotation quantity)
   (multiplier ((value 2)))
```

```
   (possessor ((cat personal-pronoun)
               (person third)
               (gender masculine)))))

(def-test t186
  "Two-fifths of the 10 cars."
  ((cat common)
   (lex "car")
   (cardinal ((value 10) (digit yes)))
   (fraction ((num 2) (den 5)))
   (definite yes)))


;; ------------------------------------------------------------
;; Zero articles for certain semantic classes:
;; ------------------------------------------------------------

(def-test t187
  "He cooks breakfast."
  ((cat clause)
   (proc ((type material) (effect-type creative) (lex "cook")))
   (partic ((agent ((cat personal-pronoun) (gender masculine)))
            (created ((cat common)
                      (lex "breakfast")
                      (denotation meal)
                      (definite yes)))))))

(def-test t188
  "He is eating dinner."
  ((cat clause)
   (proc ((type material) (lex "eat")))
   (tense present-progressive)
   (partic ((agent ((cat personal-pronoun) (gender masculine)))
            (affected ((cat common)
                       (lex "dinner")
                       (denotation meal)))))))

(def-test t189
  "He is at school."
  ((cat clause)
   (proc ((type locative)))
   (partic ((located ((cat personal-pronoun) (gender masculine)))
            (location ((cat pp) (prep ((lex "at")))
                       (np ((lex "school")
                            (cat common)
                            (denotation institution)))))))))

(store-verbs '(("build" "builds" "built" "building" "built")))

(def-test t190
  "He built the school."
  ;; Note this is not an institution but a building -> use "the"
  ((cat clause)
   (proc ((type material) (effect-type creative) (lex "build")))
   (tense past)
   (partic ((agent ((cat personal-pronoun) (gender masculine)))
            (created ((cat common)
                      (lex "school")
                      (definite yes)))))))


;; ------------------------------------------------------------
;; Determiners with pronouns
;; ------------------------------------------------------------

(def-test t203
  "He."
  ((cat personal-pronoun)
```

```
   (person third)
   (number singular)
   (gender masculine)))

(def-test t204
  "They."
  ((cat personal-pronoun)
   (person third)
   (number plural)
   (gender masculine)))

(def-test t212
  "What."
  ((cat question-pronoun)))


(def-test t191
  "All of me."
  ((cat personal-pronoun)
   (reference-number singular)
   (total +)
   (partitive yes)
   (person first)))

(def-test t192
  "You both."
  ((cat personal-pronoun)
   (total +)
   (person second)
   (number dual)))


(def-test t193
  ("The many cars."
   "The several cars.")
  ((cat common)
   (lex "car")
   (exact no)
   (number plural)
   (definite yes)
   (degree +)
   (orientation +)))


(def-test t205
  "Enough of him."
  ((cat personal-pronoun)
   (countable no)
   (person third)
   (gender masculine)
   (evaluative yes)))



;; ------------------------------------------------------------
;; Argumentative determiners: Few/many/most
;; ------------------------------------------------------------

(def-test t194
  "The few cars."
  ((cat common)
   (lex "car")
   (exact no)
   (definite yes)
   (number plural)
   (orientation -)))

(def-test t195
```

```
  ("A great many cars."
   "A good many cars."
   "Several cars."
   "Many cars.")
  ((cat common)
   (lex "car")
   (definite no)
   (number plural)
   (degree +)
   (orientation +)))

(def-test t196
  "The more cars."
  ((cat common)
   (lex "car")
   (definite yes)
   (number plural)
   (orientation +)
   (comparative yes)))

(def-test t197
  "The most cars."
  ((cat common)
   (lex "car")
   (definite yes)
   (number plural)
   (orientation +)
   (superlative yes)))

(def-test t198
  "Enough cars."
  ((cat common)
   (lex "car")
   (definite no)
   (number plural)
   (evaluative yes)
   (evaluation +)
   (orientation +)))

(def-test t199
  "Too many cars."
  ((cat common)
   (lex "car")
   (definite no)
   (number plural)
   (evaluative yes)
   (evaluation -)
   (orientation +)))

(def-test t200
  "Some butter."
  ((cat common)
   (countable no)
   (lex "butter")
   (definite no)
   (orientation +)))

(def-test t200bis
  ("A lot of butter."
   "Lots of butter."
   "Much butter."
   "Plenty of butter.")
  ((cat common)
   (countable no)
   (lex "butter")
   (definite no)
   (degree +)
   (orientation +)))
```

```
(def-test t200ter
  "A little butter."
  ((cat common)
   (countable no)
   (lex "butter")
   (definite no)
   (degree -)
   (orientation +)))



;; ------------------------------------------------------------
;; Determiners with proper nouns
;; ------------------------------------------------------------

(def-test t201
  "John."
  ((cat proper)
   (lex "John")))

(def-test t202
  "The Johns."
  ((cat proper)
   (number plural)
   (lex "John")))


;; ------------------------------------------------------------
;; Status (deictic2)
;; ------------------------------------------------------------

(def-test t206
  ("The same two individuals."
   "The identical two individuals.")
  ((cat common)
   (lex "individual")
   (cardinal ((value 2)))
   (status same)))


(def-test t207
  ("The other two individuals."
   "The different two individuals.")
  ((cat common)
   (lex "individual")
   (cardinal ((value 2)))
   (status different)))

(def-test t208
  ("The above chapter."
   "The aforementioned chapter."
   "The given chapter.")
  ((cat common)
   (lex "chapter")
   (status mentioned)))

(def-test t209
  ("This particular spice."
   "This certain spice."
   "This specific spice."
   "This original spice."
   "This special spice.")
  ((cat common)
   (lex "spice")
   (distance near)
   (status specific)))

(def-test t210
```

```
  ("His usual walk."
   "His well-known walk."
   "His normal walk."
   "His typical walk."
   "His habitual walk."
   "His expected walk.")
  ((cat common)
   (lex "walk")
   (possessor ((cat personal-pronoun)
               (person third) (gender masculine)))
   (status usual)))

(def-test t211
  ("His entire life."
   "His whole life."
   "His complete life.")
  ((cat common)
   (lex "life")
   (possessor ((cat personal-pronoun)
               (person third) (gender masculine)))
   (status entire)))
```

Appendix C
Paraphrasing power 1: NP-planning

This appendix demonstrates the level of paraphrasing power gained by including NP planning within the lexical chooser. The task of NP planning is discussed in Sect.6.5.3 (p.180) and Sect.6.5.4 (p.183). It consists of taking a conceptual set description, and mapping it first to a high-level structure where various elements of the set description are mapped to the NP head, determiner sequence and modifiers. In a second stage, each modifier is mapped to an appropriate type of syntactic modifier: describer, classifier, PP or relative qualifier.

In the first stage, the selection of a high-level NP structure is controlled by the following four features:

- realize-extension: yes or no.
- realize-intension: yes, no or clause.
- realize-reference: yes, no or clause.
- realize-quantity: evaluation, cardinal or no.

This defines a set of (2x3x3x3)=54 modes of realization for a fully specified set. Of these, only 30 express sufficient information from the set description. In this appendix, the various realizations obtained from the same set description by varying these features are listed. The set is defined as follows:

```
Intension: Topic is interesting
Extension: NLP, Vision and Robotics
Reference: Topic is covered in AI
Reference cardinal: 10
```

The various realizations are shown below, with the relevant NP highlighted in italics:

John wants to learn about *many interesting topics covered in AI, such as Vision, NLP & Robotics.*

John wants to learn about *many interesting topics such as Vision, NLP and Robotics.*

John wants to learn about *three interesting topics: Vision, NLP and Robotics.*

John wants to learn about *interesting topics such as Vision, NLP and Robotics.*

AI covers *many interesting topics such as Vision, NLP and Robotics.*

AI covers *three interesting topics: Vision, NLP and Robotics.*

AI covers *interesting topics: Vision, NLP and Robotics.*

John wants to learn about *many interesting topics covered in AI.*

John wants to learn about *three interesting topics covered in AI.*

John wants to learn about *interesting topics covered in AI.*

AI covers *many interesting topics.*

AI covers *three interesting topics.*

AI covers *interesting topics.*

John wants to learn about *many topics covered in AI such as Vision, NLP and Robotics.*

John wants to learn about *Vision, NLP and Robotics.*

Many topics covered in AI, such as Vision, NLP and Robotics, are interesting.

Vision, NLP and Robotics are interesting. [No Quantity]

Many topics covered in AI are interesting.

Three topics covered in AI are interesting.

The input FDs for each of these sentences, before lexical choice, are shown below. The set description appears under the attribute `topic` in each of the examples. The three embedding clauses shown in the previous examples are listed. To produce the various forms, only the four realize-XXX features are changed.

```
(def-test d1
  "John wants to learn about many interesting topics
   covered in AI, such as NLP, Vision and Robotics."
  ((cat clause)
   ;; Point to the non-lexicalized elements if the rest
   ;; of the clause is already lexicalized.
   (lex-cset ((+ {lex-roles soa lex-roles topic})))
   (process ((lex "want") (type lexical)
             (subcat ((1 {^3 lex-roles influence})
                      (2 {^3 lex-roles soa})
                      (2 ((cat clause)
                          (mood infinitive)
                          (oblique ((1 ((index {^4 1 index})
                                        (gap yes)))))))))))
   (lex-roles
    ((influence ((cat proper) (lex "John")))
     (soa
      ((proc ((type lexical) (lex "learn")
              (subcat ((2 ((cat pp)
                           (prep ((lex "about")))
                           (np {^4 lex-roles topic})))))))
       (lex-roles
        ((topic
          ((kind ((cat topic) (typical-size 3)))
           (semr
            ((cat set)

             ;; Realization constraints
             (realize-intension yes)
             (realize-extension yes)
             (realize-reference yes)
             (realize-quantity evaluation)

             ;; Truth conditions
             (extension ~( ((semr ((cat topic) (name nlp))))
                           ((semr ((cat topic) (name vision))))
                           ((semr ((cat topic) (name robotics)))) ))
             (cardinality 3)
             (reference
              ((cat set) (cardinality 10)
               (intension
                ((process ((semr ((cat course-relation)
                                  (name topics)))))
                 (roles
                  ((class ((semr ((cat class) (name ai-class)))))
                   (topics ((semr ((cat topic)
                                   (name generic)))))))
                 (argument {^ roles topics})))))
             (intension
              ((cat clause)
               (process ((type ascriptive)))
               (participants
                ((attribute ((semr ((cat judgement)
                                    (name interest)))))
                 (carrier ((semr ((cat topic)
                                  (name generic)))))))
               (argument {^ participants carrier})))))))))))))


(def-test d3
  "AI covers many interesting topics such NLP, Vision and Robotics."
  ((process ((semr ((cat course-relation) (name topics)))))
   (roles
    ((class ((semr ((cat class) (name ai-class)))))
     (topics
      ((semr
        ((cat set)
```

```
          ;; Realization constraints
          (realize-intension yes)
          (realize-extension yes)
          (realize-reference clause)
          (realize-quantity evaluation)

          ;; Truth conditions
          (extension ~( ((semr ((cat topic) (name nlp))))
                        ((semr ((cat topic) (name vision))))
                        ((semr ((cat topic) (name robotics)))) ))
          (cardinality 3)
          (reference
           ((cat set)
            (cardinality 10)
            (intension
             ((process ((semr ((cat course-relation) (name topics)))))
              (roles
               ((class ((semr ((cat class) (name ai-class)))))
                (topics ((semr ((cat topic) (name generic)))))))
              (argument {^ roles topics})))))
          (intension
           ((cat clause)
            (process ((type ascriptive)))
            (participants
             ((attribute ((semr ((cat judgement) (name interest)))))
              (carrier ((semr ((cat topic) (name generic)))))))
            (argument {^ participants carrier}))))))))))))


(def-test d13
  "Many topics covered in AI, such as Vision, NLP and Robotics,
   are interesting."
  ((process ((type ascriptive)))
   (cat clause)
   (lex-cset ((+ {participants attribute} {participants carrier})))
   (participants
    ((attribute ((semr ((cat judgement) (name interest)))))
     (carrier
      ((kind ((cat topic) (typical-size 3)))
       (semr
        ((cat set)

         ;; Realization constraints
         (realize-intension clause)
         (realize-extension yes)
         (realize-reference yes)
         (realize-quantity evaluation)

         ;; Truth conditions
         (extension ~( ((semr ((cat topic) (name nlp))))
                       ((semr ((cat topic) (name vision))))
                       ((semr ((cat topic) (name robotics)))) ))
         (cardinality 3)
         (reference
          ((cat set)
           (cardinality 10)
           (intension
            ((process ((semr ((cat course-relation) (name topics)))))
             (roles
              ((class ((semr ((cat class) (name ai-class)))))
               (topics ((semr ((cat topic) (name generic)))))))
             (argument {^ roles topics})))))
         (intension
          ((cat clause)
           (process ((type ascriptive)))
           (participants
            ((attribute ((semr ((cat judgement) (name interest)))))
```

```
  (carrier ((semr ((cat topic) (name generic)))))))
(argument {^ participants carrier}))))))))))))
```

Appendix D
Paraphrasing Power 2: Clause Planning

Clause planning is introduced and discussed in Sect.6.4.3 (p.170). This appendix illustrates two forms of clause planning which produce different forms of paraphrasing:

- Handling of the floating constraint of argumentation: deciding at which level in the clause an argumentative evaluation is to be realized yields different clause structures.
- Perspective selection: when the input propositional content includes more than one relation, the selection of a head relation determines the clause structure.

In the following examples, the same argumentative annotation (appearing under the AO attribute) is mapped onto different syntactic levels.

```
;; ============================================================
;; Floating aspect of argumentation
;; ============================================================

(def-test c0-1
  ("AI has many assignments."
   "AI has 6 assignments, which is a lot.")
  ((ao ((cat topos)
        (left
         ((evaluated {justification assignments})
          (realization-mode clause)
          (scale ((name "cardinal")))
          (orientation +)
          (type relative)
          (comparison-class
           ((cat set) (kind assignment)
            (reference ((name assignments)
                        (1 ((semr ((cat class)))))))
            (argument {^ reference 2})))
          (value 6)))))
   (justification
    ((rule eval-10)
     (assignments
      ((cat set) (index hw1) (kind ((cat assignment)))
       (cardinality 6)))
     (class ((cat class) (index ai1) (name ai-class)))
     (class-relation
      ((name assignments)
       (1 ((semr {justification class})))
       (2 ((semr {justification assignments}))))))))))



(def-test c0-2
  ("AI has many assignments, so it is difficult."
   "AI has 6 assignments, which is a lot, so it is difficult."
   "AI requires many assignments."
   "AI requires 6 assignments, which is a lot.")
  ((ao ((cat topos)
        (left
         ((evaluated {justification assignments})
          (realization-mode clause)
          (scale ((name "cardinal")))
          (orientation +)
          (type composite)
          (value {right})))
        (right
         ((evaluated {justification class})
          (realization-mode clause)
          (scale ((name "workload")))
          (orientation +)
          (value +)))))
   (justification
    ((rule eval-10)
     (assignments
      ((cat set) (index hw1) (kind ((cat assignment)))
       (cardinality 6)))
     (class ((cat class) (index ai1) (name ai-class)))
     (class-relation
      ((name assignments)
       (1 ((semr {justification class})))
       (2 ((semr {justification assignments}))))))))))
```

In the following examples, different perspective selections yield different clause structures for the same propositional content:

```
;; ============================================================
;; Perspective selection
;; ============================================================

(def-test c1
  ("AI requires one assignment in which you do not have experience."
   "You do not have experience in one assignment which
    Intro to AI requires."
   "One assignment which Intro to AI requires consists of a paper.")

  ;; Argumentative orientation: chain
  ;; /+ assignments, + workload/, /+ workload, + difficult/,
  ;; /+ difficult, -take/
  ((ao ((cat topos)
        (left
         ((evaluated {justification assignments})
          (scale ((name "cardinal")))
          (orientation +)
          (type composite)
          (value {^2 chain1 right}))))
        (chain1 ((cat topos)
                 (left {ao left})
                 (right
                  ((evaluated {justification class})
                   (scale ((name "workload")))
                   (value +)
                   (orientation +)))))
        (chain2 ((cat topos)
                 (left {ao chain1 right})
                 (right ((evaluated {^2 left evaluated})
                         (scale ((name "difficult")))
                         (orientation +)))))
        (chain3 ((cat topos)
                 (left {ao chain2 right})
                 (right {ao right})))
        (right ((cat evaluation)
                (evaluated {justification class})
                (scale ((name "take")))
                (orientation -)))))

   ;; Propositional content:
   ;; assignments(class, HWSet)
   ;; hw-type(HWSet, paper)
   ;; experience(hearer, paper)
   (justification
    ((rule eval-12)
     (assignments
      ((cat set) (index hw2) (kind ((cat assignment))) (cardinality 1)
       (extension
        ((car ((semr ((cat assignment) (name paper1))))) (cdr none)))))
     (class ((cat class) (index ai1) (name ai-class)))
     (hw-activities
      ((cat set) (index hwa1) (kind ((cat hw-activity)))
       (cardinality 1)
       (realize-extension yes)
       (extension
        ((car ((semr ((cat hw-activity) (name paper1)))))
         (cdr none)))))
     (class-relation
      ((name assignments)
       (1 ((semr {justification class})))
       (2 ((semr {justification assignments})))))
     (user-relation
      ((name experience) (orientation -)
       (1 ((semr ((cat student) (name hearer)))))
       (2 ((semr {justification hw-activities})))))
     (object-relation
      ((name hw-type)
```

```
       (1 ((semr {justification assignments})))
       (2 ((semr {justification hw-activities}))))))))))


(def-test c4
  ("Intro to AI has many assignments which consist of programming."
   "You have experience in programming which many
    AI assignments require."
   "Many AI assignments consist of programming.")

  ((ao ((cat topos)
        (left
         ((evaluated {justification assignments})
          (scale ((name "cardinal")))
          (orientation +)
          (value {right})
          (type composite)))
        (chain1 ((cat topos)
                 (left {ao left})
                 (right
                  ((evaluated {justification class})
                   (realization-mode clause)
                   (scale ((name "programming")))
                   (value +)
                   (orientation +)))))
        ;; /+ programming, + interest/
        ;; This topos is only valid for a certain class of users, as
        ;; indicated by the (um yes) feature
        (chain2 ((cat topos) (left {ao chain1 right})
                 (right ((evaluated {justification class})
                         (realization-mode clause)
                         (scale ((name "interest")))
                         (orientation +)
                         (um yes)
                         (value +)))))
        (chain3 ((cat topos)
                 (left {ao chain2 right})
                 (right {ao right})))
        (right ((cat evaluation)
                (evaluated {justification class})
                (scale ((name "take")))
                (orientation +)))))

   (justification
    ((rule eval-4) (course ((cat class) (index ai1) (name ai-class)))
     (assignments ((cat set) (index hw3) (kind ((cat assignment)))
                   (cardinality 2)))
     (hw-activity ((cat hw-activity) (index hwa2) (name programming)))
     (class-relation
      ((name assignments)
       (1 ((semr {justification course})))
       (2 ((semr {justification assignments})))))
     (user-relation
      ((name experience) (orientation +)
       (1 ((semr ((cat student) (name hearer)))))
       (2 ((semr {justification hw-activity})))))
     (object-relation
      ((name hw-type)
       (1 ((semr {justification assignments})))
       (2 ((semr {justification hw-activity}))))))))))
```

Appendix E
Example of ADVISOR II Output

This appendix shows examples of output produced by the ADVISOR II system. As discussed in Chap.7 (p.211), ADVISOR II can answer three types of queries:

- How <att> is <c>?
- Should I take <c>?
- How is <c>?

where <c> is a class and <att> is an attribute of a class, for example *difficult* or *workload*. ADVISOR II relies on a user model indicating which courses the user has taken, his or her interests and skills.

ADVISOR II is not a complete interactive system: it requires queries to be typed as Lisp functions and it requires user intervention during paragraph planning. The two main functions of ADVISOR II are:

- (evaluate <course> <student> <scale>)
- (evaluate-all-scales <course> <student>)

A question *Should I take <c>* is formulated as a call to `(evaluate <c> <current-student> "take")` - where the scale "take" determines how much a course is appropriate to a given student. `Evaluate-all-scales` evaluates the course on all the known scales of the system and reports the list of activated scales for the given course/user model combination. It corresponds to a query of the type *how is <c>*.

Courses and students are stored and described as CLASSIC instances [Resnick et al. 90]. In the following examples, the following user models are used:

```
;; A junior student concentrating on AI with a ‘‘hacker’’ tendency.
(cl-create-ind ’user1 ’(and student
                            (fills degree-candidacy bs)
                            (fills year junior)
                            (fills semester f91)
                            (fills points 30)
                            (fills taken intro-class1
                                   data-structures-class1
                                   discrete-math-class1)
                            (fills concentration ai)
                            (fills profile+ programming)
                            (fills profile- math paper1)
                            (fills interest ai nlp)))


;; A sophomore student concentrating on theory - mainly mathematical
;; background.
(cl-create-ind ’user2 ’(and student
                            (fills degree-candidacy bs)
                            (fills year sophomore)
                            (fills semester f91)
                            (fills points 30)
                            (fills taken intro-class2
                                   data-structures-class2)
                            (fills concentration theory)
                            (fills profile- programming paper1)
                            (fills profile+ math proof1)
                            (fills interest theory complexity algorithm)))

;; A senior student concentrating on hardware
(cl-create-ind ’user3 ’(and student
                            (fills degree-candidacy bs)
                            (fills year senior)
                            (fills semester f91)
                            (fills points 30)
                            (fills taken intro-class3
                                   data-structures-class3
                                   discrete-math-class3
                                   graph-theory-class3
                                   circuits-class3
                                   os-class3)
                            (fills concentration hardware)
                            (fills profile- proof1 paper1 math)
                            (fills profile+ programming)
                            (fills interest hardware circuits vlsi)))
```

Note that in these examples, role fillers are generally other CLASSIC entities defined with additional information. For example, topics are described as follows:

```
(cl-create-ind ’algorithm ’(and topic (fills area theory software)))
(cl-create-ind ’expert-systems ’(and topic (fills area ai)))
(cl-create-ind ’vlsi ’(and topic (fills area hardware)))
(cl-create-ind ’parsing ’(and topic (fills area software ai)))
...
```

Courses are described according to the following schema:

```
(cl-create-ind 'ai-class '(and class
                               (fills area ai)
                               (fills prereq data-structures-class)
                               (fills normal-seq junior)
                               (fills topic ai expert-systems vision nlp
                                      search robotics game-playing logic)
                               (fills required yes)
                               (fills followup vision-class nlp-class
                                      expert-system-class kr-class
                                      robotics-class)
                               (fills assignment
                                      prog1 prog2 prog3 paper1)))


(cl-create-ind 'data-structures-class
               '(and class
                     (fills area software)
                     (fills prereq intro-class)
                     (fills normal-seq freshman)
                     (fills topic software programming algorithm
                            data-structures)
                     (fills required yes)
                     (fills followup ai-class analysis-alg-class os-class)
                     (fills assignment prog1 prog2 prog3 prog4 prog5 prog6)))
```

The different types of assignments known to the system are:

- paper
- programming
- presentation
- proof
- project
- lab

The courses are evaluated on the following scales:

- take
- interest
- importance
- level
- difficulty
- workload
- programming
- math

The system is organized as follows: the evaluation system passes the input pair course/student to all existing evaluation functions. The output of all the functions that fire is then passed to the topoi base and chaining is performed to link the observations triggered by the evaluation functions to the input evaluation scale. The list of evaluation chains is then filtered with human intervention and a paragraph plan is built and passed to the lexical chooser and syntactic realization grammar.

In the following journal of execution, all 13 existing evaluation functions and 12 existing topoi are exercised at least once in the paragraph planner. All corresponding knowledge base relations are therefore exercised at least once in the lexical chooser. Both evaluation functions and topoi are listed in Chap.7.

Running time is typically 4mns on a 1991 Sparcstation model with 16Mb of RAM, running Lucid Common Lisp. Running time is broken down as follows:

- 30s for the content planner (including evaluation functions, topoi chaining, chain merging and hierarchy formation);
- 2:30mn for lexical choice;
- 1mn for syntactic realization.

A journal of execution follows:

```
USER 3> (load "advisor")

; Loading ~elhadad/advisor/advisor.l.

;; ============================================================
;; LOADING FUF
;; ============================================================
; Fast loading /users/black4/segel/elhadad/fuf/freeze/fug5.fasl.
; Loading macros.l.
; Fast loading vars.fasl.
; Fast loading trace.fasl.
; Fast loading generator.fasl.
; Fast loading wait.fasl.
; Fast loading define.fasl.
; Fast loading backtrack.fasl.
; Fast loading external.fasl.
; Fast loading determine.fasl.
; Fast loading type.fasl.
; Fast loading ignore.fasl.
; Fast loading alt.fasl.
; Fast loading ralt.fasl.
; Fast loading fset.fasl.
; Fast loading control.fasl.
; Fast loading graph.fasl.
; Fast loading pattern.fasl.
; Fast loading path.fasl.
; Fast loading top.fasl.
; Fast loading lexicon.fasl.
; Fast loading linearize.fasl.
; Fast loading checker.fasl.
; Fast loading fdlist.fasl.
; Fast loading complexity.fasl.
; Fast loading continue.fasl.
; Fast loading test.fasl.

    FUF Version 5.1, Copyright (C) 1987-1992 Michael Elhadad.
    FUF comes with ABSOLUTELY NO WARRANTY; for details type (fug5::warranty).
    This is free software, and you are welcome to redistribute it
    under certain conditions, type (fug5::license) for details.

;; ============================================================
;; LOADING SYNTACTIC REALIZATION GRAMMAR
;; ============================================================
; Loading ~elhadad/grammar/gr.l.
; Loading tpat.l.
; Loading types.l.
; Loading clause.l.
; Loading transitivity.l.
; Loading voice.l.
; Loading circumstance.l.
; Loading verb-group.l.
; Loading np.l.
; Loading determiner.l.
; Loading complex.l.
; Loading gr-modular.l.

;; ============================================================
;; LOADING LEXICAL CHOOSER
;; ============================================================
; Loading ~elhadad/grammar/lex.l.
; Loading set2.l.
; Loading sentence-planning.l.
; Loading ao-planning.l.
; Loading connectives.l.
; Loading topoi.l.
; Loading lexicalize4.l.

; Loading ~elhadad/advisor/kb/kb.l.
```

```
;; ============================================================
;; LOADING CLASSIC
;; ============================================================
; Loading ~elhadad/classic/classic.l.
; Fast loading load.fasl.
; Fast loading lisp-util.fasl.
; Fast loading error.fasl.
; Fast loading fc.fasl.
; Fast loading nc.fasl.
; Fast loading rr.fasl.
; Fast loading ind.fasl.
; Fast loading role.fasl.
; Fast loading table.fasl.
; Fast loading partition.fasl.
; Fast loading util.fasl.
; Fast loading user.fasl.
; Fast loading graph.fasl.
; Fast loading prop.fasl.
; Fast loading ccl.fasl.
; Fast loading icl.fasl.
; Fast loading norm-ind.fasl.
; Fast loading norm.fasl.
; Fast loading norm-eqns.fasl.
; Fast loading read.fasl.
; Fast loading update.fasl.
; Fast loading host.fasl.
; Fast loading glcil.fasl.
; Fast loading cl-unix.fasl.
; Fast loading init.fasl.

CLASSIC Version 1.1

** Type (cl-print-version-info) or (cl-print-updates) to see
** the updates to the previous version.

;; ============================================================
;; Loading KNOWLEDGE BASE
;; ============================================================
; Fast loading ~elhadad/advisor/kb/kb-interface.fasl.
; Fast loading upper-model.fasl.
; Fast loading domain.fasl.
; Fast loading instances.fasl.


;; ============================================================
;; Loading EVALUATION SYSTEM AND PARAGRAPH PLANNER
;; ============================================================
; Fast loading ~elhadad/advisor/eval/eval-functions.fasl.
; Fast loading ~elhadad/advisor/eval/topoi.fasl.

T


USER 4> (in-package "ADVISOR")

;; ============================================================
;; The following examples cover all evaluation functions and all
;; topoi links.
;; ============================================================


;; ============================================================
;; FIRST EXAMPLE: SHOULD USER3 TAKE NETWORKS?
;; ============================================================
;; Note that the scale argument is "take" by default.
ADVISOR 5> (evaluate networks-class user3)

Please select 2 chains out of the following,
```

```
starting with the conclusion you want to support:

0: EVAL-7 => -math => -interest => -good
1: EVAL-1 => +programming => -interest => -good

Enter two numbers: 0 1

;; ================================================
;; Starting Class NETWORKS-CLASS - User USER3
;; ============
;; 7 (um.math+)(area <> theory) => -math => -interest => -good
;; 1 (um.pgm-)(asg.pgm>0) => +programming => -interest => -good

Your concentration is in theory and Networks is not a theory course.
In addition, Networks has many programming assignments, so it involves
a lot of programming.
Therefore it would not be interesting for you.
I would not recommend it.


ADVISOR 6> (evaluate networks-class user2)

Please select 2 chains out of the following,
starting with the conclusion you want to support:

0: EVAL-14 => +level => +difficulty => -good
1: EVAL-7 => -math => -interest => -good
2: EVAL-1 => +programming => -interest => -good

Enter two numbers: 0 1

;; ================================================
;; Starting Class NETWORKS-CLASS - User USER2
;; ============
;; 14 (um.year Y)(normal-seq>Y) => +level => +difficulty => -good
;; 7 (um.math+)(area <> theory) => -math => -interest => -good

You are a sophomore and Networks is generally taken in senior year,
so it could be quite advanced for you. Therefore, you could have
trouble with it.
Furthermore, your area is in theory, and Networks covers little math
because it is not a theory class.
So you could find it not interesting.
I would not recommend it.
```

The following examples were obtained by running an iteration program through the knowledge base, systematically combining all generated evaluation chains for the courses and users defined. The sample presented covers all missing evaluation functions and topoi.

```
;; ================================================
;; Starting Class NETWORKS-CLASS - User USER1
;; ============
;; 6 (um.math-)(topic in theory) => +math => +difficulty => -good
;; 4 (um.pgm+)(asg.pgm<4) => +programming => +interest => +good

Networks has some programming assignments, so it involves some
programming.  You would find it interesting.
But it covers queuing theory, a very theoretical topic, so it is quite
mathematical.  It could be difficult for you.
I would not recommend it.

;; ================================================
;; Starting Class GRAPHICS-CLASS - User USER1
;; ============
;; 3 (um.pgm+)(asg.pgm>3) => +programming => +interest => +good
;; 10 (asg>4) => +workload => +difficulty => -good

Graphics requires many assignments, so the workload can be quite high,
so it could be difficult.
But you are interested in programming and it has many programming
assignments, so it should interest you.
It should be a good course.

;; ================================================
;; Starting Class VLSI-CLASS - User USER2
;; ============
;; 11 (asg<3) => -workload => -difficulty => +good
;; 14 (um.year Y)(normal-seq>Y) => +level => +difficulty => -good

You are a sophomore and VLSI is usually taken in junior year,
so it could be advanced.
But it has few assignments, so the workload is quite low.
You should not have trouble with it.
It should be a good course.


;; ================================================
;; Starting Class DISCRETE-MATH-CLASS - User USER1
;; ============
;; 5 (um.math-)(area theory) => +math => +difficulty => -good
;; 2 (asg.pgm=0) => -programming => -interest => -good

Discrete math is a theory class so it is quite mathematical.  You could
find it difficult.
Furthermore Discrete Math does not deal with programming
because it has no programming assignments.
So you would not find it interesting.
I would not recommend it.


;; ================================================
;; Starting Class DATA-STRUCTURES-CLASS - User USER3
;; ============
;; 15 (followup>2) => +importance => +good
;; 9 (um.interest Y)(Y covered) => +interest => +good

Many courses use Data structures - so it is an important course.
Furthermore you should find Data structures interesting
because it covers algorithms, which interests you.
I would definitely recommend it.

;; ================================================
;; Starting Class AI-CLASS - User USER1
;; ============
;; 12 (um.asg-type-)(asg-type>0) => +workload => +difficulty => -good
;; 9 (um.interest Y)(Y covered) => +interest => +good

AI deals with many interesting topics, such as NLP, Vision and KR.
```

```
But it has many assignments which consist of writing papers.
You have little experience writing papers.
So you could have trouble with it.
I would not recommend it.

;; ================================================
;; Starting Class COMPILERS-CLASS - User USER1
;; ============
;; 14 (um.year Y)(normal-seq>Y) => +level => +difficulty => -good
;; 3 (um.pgm+)(asg.pgm>3) => +programming => +interest => +good
Although Compilers could be difficult because it is usually
taken in Senior year, it should be interesting because it has
a good many programming assignments.
It should be a good course.
```

The following examples demonstrate the *how <att> is <c>* form of queries.

```
;; ================================================
;; Starting Class AI-CLASS - User USER1 - Scale "interest"
;; ============
;; How interesting is AI?
ADVISOR 72> (evaluate ai-class user1 :scale "interest")

Please select 2 chains out of the following,
starting with the conclusion you want to support:

;; 0: EVAL-9 => +interest
;; 1: EVAL-6 => +math => -interest
;; 2: EVAL-4 => +programming => +interest


;; 0/1
You have no interest in theory and AI deals with one mathematical
topic, logic.  But AI covers a good number of interesting topics -
Expert systems, NLP, vision.  So it is quite interesting.

;; 0/2
AI deals with many topics which interest you.  In addition,
it has a good amount of programming assignments and you like
programming.  It should be interesting.

;; 1/2
You have experience in programming and AI has many assignments which
consist of programming, so it is a programming course.
But it is quite theoretical, because it covers
some very mathematical topics. You would not find it interesting.
```

Finally the following examples demonstrate the work of the paragraph planner when only one evaluation chain is considered. In all previous examples, two evaluation chains were combined.

```
;; --------------------
;; One chain output
;; --------------------
;;
;; How is the workload of OS? (user 2)
ADVISOR 120> (evaluate1 os-class user2 :scale "workload")

;; 0: EVAL-14 => +level => +difficulty => +workload
;;
OS is usually taken in junior year and you are a sophomore.
So it could be quite advanced for you.
Therefore you could have trouble with it
so the workload could be quite high.

;; 1: EVAL-10 => +workload
OS has a large number of assignments,
so the workload could be quite high.

;; 2: EVAL-1 => +programming => +workload
OS has many programming assignments so it is a programming class.
Therefore the workload could be high.

;; How mathematical is Networks?
ADVISOR 132> (evaluate1 networks-class user2 :scale "math")

;; 0: EVAL-7 => -math
Your concentration is in theory and Networks is not a theory class.
So it is not very mathematical.


;; How mathematical is Networks? (user1)
ADVISOR 153> (evaluate1 networks-class user1 :scale "math")

;; 0: EVAL-6 => +math
Networks covers queuing theory, a highly mathematical topic.
You have no experience in theory.  So it is quite mathematical.
```

Table of Contents

List of Figures