Desarrollo de un parser funcional para el lenguaje castellano

Pablo Ariel Duboué[1]

7 de Agosto, 1998

[1]El autor agradece el apoyo de la Facultad de Matemática, Astronomía y Física de Córdoba, y en particular al Dr. Javier O. Blanco quien sin su dirección este trabajo no hubiera sido posible.

Índice General

Capítulo 1

Introducción

1.1 La comunicación

El presente trabajo se encuadra dentro del campo de la *Lingüística Computacional*, por lo tanto es una aplicación de teorías que de algún modo tratan de explicar la estructura del lenguaje humano. Es sorprendente encontrar que la lingüística [3] ocupa un tiempo considerable en formular teorías para representar la estructura (y el funcionamiento) del lenguaje humano. ¿Qué es, después de todo, lo que hay que explicar? Hablar el lenguaje nativo de uno es una tarea natural y sin esfuerzo, llevada a cabo con gran velocidad y facilidad. Inclusive los niños pequeños pueden hacerlo con muy poco esfuerzo consciente. En base a esto, es muy común concluir que más allá de unas pocas reglas de gramática y pronunciación no hay ninguna otra cosa que explicar acerca del lenguaje humano.

Pero parece que hay un montón para explicar. Si nos movemos "un paso afuera" del lenguaje y lo miramos como un objeto a ser conscientemente estudiado y descripto y no meramente usado, descubrimos una excitante esfera del conocimiento humano que previamente nos estaba oculta.

1.1.1 ¿Qué es la lingüística?

El campo de la lingüística, el estudio científico del lenguaje natural humano, es un área de estudio muy interesante y en crecimiento, con importante impacto en áreas tan diversas como la educación, la antropología, la sociología, la enseñanza de idiomas, la psicología congnositiva, la filosofía, las ciencias de la computación y la inteligencia artificial, entre otras. De hecho los últimos cuatro campos mencionados, junto con la lingüística forman el campo emergente de la ciencia cognoscitiva, que estudia la estructura y el funcionamiento del proceso cognoscitivo humano.

Más allá de la importancia del campo de la lingüística, mucha gente, inclusive de muy alto nivel educativo, dirá que sólo tiene una idea vaga del objeto de estudio del campo. Algunos creen que un lingüista es una persona que habla muchos idiomas de manera fluida. Otros creen que los lingüistas son expertos que pueden ayudar a decidir cuando es correcto decir "Soy yo" o "Soy mi". Sin embargo es posible ser un lingüista profesional (y uno excelente, inclusive) sin haber enseñado una sola clase de lenguaje, sin haber traducido en la ONU y sin hablar más que un idioma.

¿Qué es la lingüística, entonces? Fundamentalmente, su materia de estudio se interesa en la naturaleza del lenguaje y la comunicación. Aparentemente la gente ha estado fascinada con el lenguaje y la comunicación desde hace miles de años, y a pesar de ello en muchos sentidos recién estamos comenzando a comprender la compleja naturaleza de este aspecto de la vida humana. Si preguntácemos, ¿Cuál es la naturaleza del lenguaje? o ¿Cómo funciona la comunicación? rápidamente nos daríamos cuenta que estas preguntas no tienen respuestas sencillas y son demasiado amplias para ser respondidas de forma directa. De manera similar, preguntas tales como ¿Qué es la energía? o ¿Qué es la materia? no pueden ser respondidas de manera simple, y de hecho todo el campo de la física es un intento por contestarlas. La lingüística no es diferente: el campo como un todo representa un intento de quebrar las amplias preguntas sobre la naturaleza del lenguaje y la comunicación en preguntas más pequeñas y manejables que esperamos se puedan responder, y al hacer esto establecer resultados que puedan permitirnos movernos más cerca de las respuestas a las preguntas más grandes. Pero a menos que limitemos nuestras aspiraciones en esta forma y nos restrinjamos a esquemas de trabajo particulares para examinar diferentes aspectos del lenguaje y la comunicación, no podemos esperar hacer progresos en responder las preguntas amplias que han fascinado a la gente por tanto tiempo.

1.1.2 ¿Por qué estudiar el lenguaje?

Hay muchas respuestas posibles [10], y por analizar algunas no se trata de desvirtuar otras o cuestionar su legitimidad. Uno puede, por ejemplo, estar simplemente fascinado con los elementos del lenguaje en sí y querer descubrir su orden y estructura, su origen en la historia o en el individuo o las formas en las cuales se los usa en el pensamiento, en la ciencia o en el arte, o en el intercambio social normal.

Otra razón para estudiar el lenguaje —una de las razones más atractivas, por cierto— es la tentación de mirar al lenguaje, según una frase tradicional, como "espejo de la mente". Y esto no significa tan sólo que los conceptos expresados y las distinciones desarrolladas en el uso normal del idioma nos dan pistas acerca de los patrones de pensamiento y del mundo del "sentido común" construido por la mente humana. De forma más intrigante, está la posibilidad de que el estudio del lenguaje nos permita descubrir principios abstractos que gobiernan su estructura y uso, principios que son universales y por necesidad biológica y no un mero accidente histórico, que derivan de características mentales de las especies. El lenguaje humano es un sistema de remarcable complejidad. El conseguir adquirir un lenguaje humano puede llegar a ser una tarea de increible logro intelectual para una criatura no específicamente diseñada para cumplir con dicha tarea (e.g., una computadora). Y un niño normal adquiere este conocimiento bajo una relativamente poca exposición y sin entrenamiento específico. Puede, entonces, casi sin esfuerzo hacer uso de una intrincada estructura de reglas específicas y principios de guía para expresar sus pensamientos y sentimientos a otros, despertando en ellos nuevas ideas y sutiles percepciones y conclusiones. Para la mente consciente, no específicamente diseñada para este propósito, es todavía un objetivo distante reconstruir y comprender lo que para un niño es logrado intuitivamente y con un esfuerzo mínimo. Es por ello que el lenguaje es un espejo de la mente en un sentido profundo y significativo. Es un producto de la inteligencia humana, recreado nuevamente en cada individuo por operaciones que están más allá del alcance de la consciencia y el entendimiento.

1.1.3 Subáreas de la lingüística

Al estudiar las propiedades estructurales del lenguaje humano, es útil notar un detalle: el análisis estructural del lenguaje humano puede establecerse en función de: (1) unidades discretas de distintos tipos y (2) reglas y principios que gobiernan la forma en que estas unidades discretas pueden ser combinadas y ordenadas. Según las unidades mínimas analizadas, la lingüística se puede dividir en:

Morfología que se dedica al estudio de la unidad fundamental de la estructura lingüística: la palabra (Teniendo en cuenta que según [27] una persona conoce alrededor de 80.000 palabras a la edad de 17 años). Pero el dominar un lenguaje no implica solamente dominar una lista impresionantemente larga de palabras de dicho idioma (el *lexicón*) sino también comprender y poder utilizar toda una serie de **reglas** referidas a su formación y descomposición. Podríamos decir que la morfología trata de responder preguntas tales como: ¿Cuáles son las palabras y cómo se forman?, ¿Cómo se forman las palabras más complejas a partir de partes más simples? ¿Cómo se relaciona el significado de una palabra compleja respecto del significado de sus partes más simples?, ¿Como están relacionadas las palabras individuales del lenguaje con otras palabras del mismo idioma?

Fonética que se ocupa de cómo los sonidos del habla son producidos (articulados) en el tracto vocal (en un campo de estudio conocido como *fonética articulatoria*), así también como de las propiedades físicas de las ondas de sonido generadas por el tracto vocal (en otro campo que tiene por nombre *fonética acústica*).

Fonología en la cual se analizan también los sonidos del habla pero haciendo énfasis en la estructura y el uso sistemático de los sonidos en el lenguaje humano. Por un lado estudia la descripción de los sonidos presentes en un lenguaje en particular y las reglas que determinan la distribución de dichos sonidos. Y por otro, se refiere a toda una teoría general sobre propiedades universales de los sistemas de sonido de los lenguajes naturales (esto es, propiedades reflejadas en muchos, si no todos, los lenguajes humanos).

Sintaxis que pasa su centro de atención de las *palabras* (en definitiva, objeto de estudio de la morfología, la fonética y la fonología) al análisis de estructuras más grandes: *frases y oraciones*[1]. Como dentro de este tópico se centra el presente trabajo se analizará la sintaxis en más detalle en un próximo apartado. Por el momento es suficiente notar el hecho que un hablante nativo de cualquier lenguaje puede producir y comprender una cantidad infinita de oraciones de dicho lenguaje, muchas de las cuales nunca había oido o producido antes. Esto pone de manifiesto la existencia

[1] Se utiliza el término "palabra" y "oración" sin definir pues la bibliografía pertinente esta repleta de definiciones y contradefiniciones y ninguna encaja con la intuición.

de reglas inherentes para la generación y comprensión de dichas oraciones (pues la mera memorización no es suficiente).

Semántica pues el estudio de la lingüística no estaría completo si en las unidades y principios de combinación no se tuviera en cuenta lo que dichas unidades significan, a que suelen referirse, y que suelen comunicar. De este modo en el campo de la lingüística la semántica es considerada en general el estudio del *significado* (y nociones relacionadas) en los lenguajes, mientras que en el campo de la lógica, la semántica está generalmente considerada como el estudio del referente lingüístico o *denotación* o *condiciones de verdad* en los lenguajes.

1.2 Conceptos básicos de sintaxis

Para la realización del presente trabajo se analizaron tres teorías sintácticas [34] contemporáneas, las cuales comparten una cierta tradición junto con un conjunto fundamental de conceptos y de terminología. Este apartado presenta un telón de fondo sobre el que se situa después la discusión sobre dichas teorías y se ocupa del vocabulario básico y de los supuestos comunes a toda teoría sintáctica actual.

En rigor, puede que la misma idea de sintaxis tal como aquí se concibe llegue a resultar un tanto extraña, pues las distintas reglas sintácticas o gramaticales que solemos recordar de la escuela secundaria son más bien de naturaleza prescriptiva. Por ejemplo decir *"fui a lo de mi tía"* y no *"fui de mi tía"*. Los factores que gobiernan uno u otro uso son de naturaleza histórica en parte y, en nuestro ambiente actual, también sociológicos. Desde el punto de vista sintáctico —en el sentido técnico—, las fuerzas de la historia y de la sociología carecen casi de importancia pues la la sintaxis no se ocupa de la distinción de una frase del castellano como "fui a lo de mi tía" y una variante tolerable (si bien no aconsejada). La sintaxis se ocupa, en cambio, de las frases posibles e imposibles del castellano o cualquier otra lengua natural. La clave estriba en la diferencia de aceptación por parte del hablante nativo, esto es, en la certeza de que una oración A es posible y que otra oración B no resulta aceptable, por muchas vueltas que se le dé, en la lengua en cuestión. Si alguien desarrolla una teoría de la gravedad, encontrará que en lo esencial la cosas se caen y rara vez se elevan. La diferencia entre oraciones posibles e imposibles no se distingue en principio de cuando se construye una teoría para explicar un cierto comportamiento observado, como la diferencia entre la atracción hacia un cuerpo más grande y su repulsión; y en la práctica aquella diferencia lingüística es al menos tan difícil de discernir como la "caída pura", es decir la abstracción hecha a partir de los efectos y perturbaciones externas. En el apartado siguiente se analizará la naturaleza de los datos sintácticos tal como los conciben quienes elaboran las teorías.

No obstante, el estudio de la sintaxis no está necesariamente divorciado de consideraciones quizá más tangibles de historia, sociología y de otras cosas que contribuyen a formar la experiencia humana. En último término, cuando sepamos mucho más sobre todas estas cosas, la sintaxis ocupará su lugar en el ámbito de teorías más amplias sobre la acción y la interacción humanas. Pero hasta entonces parece razonable dejar a un lado estas consideraciones más amplias. Uno de los caracteres más convincentes de la sintaxis es que mientras su estudio constituye una empresa teórica tremendamente abstracta, los estudiosos descubren y mejoran su conocimiento sobre fenómentos que son supuestamente bien reales.

1.2.1 Competencia sintáctica

El punto de partida para comprender la sintaxis en su concpeción actual se encuentra en una pregunta del tipo "¿Qué sabemos cuando sabemos una lengua (por ejemplo el castellano)?" La respuesta que emitió Noam Chomsky en su libro *Syntactic Structures* [9] en 1957 dió lugar a una disciplina enteramente nueva en el seno de la lingüística. Su respuesta decía que lo que sabemos es una colección de palabras y de reglas con las que **generamos** cadenas de esas palabras que llamamos oraciones de nuestra lengua. Más aun, a pesar de que hay un número finito de elementos en aquella colección —digamos, varios miles de palabras y no más de unos cuantos centenares de reglas—, puede, sin embargo, generarse un número infinito de oraciones, a veces muy largas. Ello se debe a que algunas de las reglas son recursivas. La rama de la lingüística que adopta este punto de vista, de que lo importante es el modo de generar oraciones, se conoce como **Gramática Generativa**.

1.2.2 Gramaticalidad

La lengua expresa en último término una relación entre un sonido y un significado que se encuentran en los extremos opuestos del espectro lingüístico. Los sonidos que han sido producidos para este trabajo —o, más bien, símbolos gráficos— permiten que se extraiga de ellos un significado. En alguna parte hacia la mitad de este proceso reside la sintaxis. Así, por ejemplo, la oración *La mujer vendió las pinturas* puede contener un significado que *Vendió las pinturas mujer la* no puede. Dos aspectos interesan aquí: ante todo, no importa que no podamos asegurar qué viene a significar en último término este segundo ejemplo, aunque seguramente se adivine; la cuestión es que no se habla castellano si se elige esta forma de expresar tal significado. En segundo lugar, hay muchas lenguas en el mundo en las que la traducción palabra por palabra del segundo ejemplo sería un modo efectivamente gramatical (y en algunos casos el único) de distribuir las palabras en cuestión para expresar dicho significado. Así, pues, no hay nada inherente al propio mensaje que explique por sí mismo por qué es imposible que aquella secuencia sea una oración del castellano.

En [9], Chomsky adujo el ejemplo (1.1) ya célebre para ilustrar la importancia que tiene investigar la sintaxis independientemente (al menos en parte) de otras consideraciones:

(1.1)

Colorless green ideas sleep furiously.

Aunque se trata de un texto sin ningún sentido corriente, es sintácticamente impecable. Por el contrario, el siguiente ejemplo carece de sentido y es además aberrante desde el punto de vista sintáctico:

(1.2)

*Furiously sleep ideas green colorless.

El asterisco antepuesto indica que se trata de un ejemplo sintácticamente mal formado. De ahí que debamos disponer de unas reglas que produzcan (1.1), pero no (1.2). De nuevo observamos la distinción sintáctica si sustituimos las palabras de estos ejemplo a fin de producir un resultado con sentido:

(1.3)
(a) Revolutionary new ideas appear infrequently.

(b) *Infrequently appear ideas new revolutionary.

En estos ejemplos se ha cambiado algunas palabras por otras que pertenecen a la misma parte de la oración o, como se dice en un uso lingüístico más habitual, a la misma **categoría sintáctica**. En rigor, este cambio muestra otro aspecto de la sintaxis destacado por Chomsky, el de la estructura de las oraciones, pues la auténtica realidad sintáctica que subyace a los contrastes de gramaticalidad que hemos visto antes reside en que mientras hay secuencias formadas por

Adjetivo-Adjetivo-Nombre-Verbo-Adverbio

que son gramaticales en inglés, otras secuencias formadas por

Adverbio-Verbo-Nombre-Adjetivo-Adjetivo

no lo son. Conectando ciertas combinaciones de palabras puede llegarse también a oraciones significativas en el primer caso, pero esto no tiene nada que ver con la buena formación sintáctica. Aunque aquí se ha hablado de cadenas de palabras, el aspecto más importante de la sintaxis es la estructura de dichas cadenas y no ellas por sí mismas.

1.2.3 Propiedades universales del lenguaje

Uno de los fines de la teoría sintáctica consiste en proporcionar un ámbito descriptivo dentro del cual quede prevista con precisión la gama de variaciones que cabe encontrar de una a otra lengua. Es decir, nos interesa disponer de una teoría suficientemente flexible que sea capaz de caracterizar las variaciones más sutiles que cabe encontrar, pero aun así que impida incluso tomar en consideración otras determinadas posibilidades. Para mencionar un ejemplo común de esto último, digamos que no hay lenguas en que las preguntas se formen a partir de oraciones normales mediante la inversión de todas las palabras de la oración, como si (1.3)(b), de más arriba, fuese la versión interrogativa de (1.3)(a).

Volviendo a apartados anteriores, en última instancia este tipo de saber nos llevaría hasta campos inéditos de investigación en torno a la mente. Cabría entonces preguntarse algo así como "¿Qué podemos averiguar sobre los procedimientos y capacidades computacionales del cerebro sabiendo que nunca debe actuar para invertir todos los símbolos lingüísticos de una secuencia dada?"

o bien "¿Cómo es posible que se aprenda la lengua?" Se considera que esta última pregunta reviste una gran importancia, ya que ha servido para configurar programas enteros de investigación en sintaxis. En efecto, Chomsky concibe la lingüística (es decir, la sintaxis) como una parte de la psicología cognoscitiva, ya que, en último término, lo que hacemos es examinar las propiedades de la mente. Existe, por cierto, una amplia y rozagante subdisciplina, la psicolingüística, que vincula lo que sabemos de la mente con el lenguaje.

Sin embargo, los análisis que aquí se presentan quedan básicamente al margen de la psicolingüística (aunque el término "universal" se refiere a propiedades que en última instancia se atribuyen a fenómenos mentales y no, por ejemplo, sociales). Queda a discreción de cada cual decidir si una teoría sintáctica debe considerarse como una muestra de directa de nuestro saber lingüístico o como la descripción de un sistema cuyo comportamiento modela simplemente nuestro comportamiento lingüístico.

1.2.4 Los datos

Un detalle importante de la investigación lingüística es el siguiente: Toda distinción de estructura sintáctica que encontramos resulta de un pequeño experimento que consiste en encontrar un hablante nativo de irlandés, hopi o lo que sea y tratar de intervenir en su sentido de la gramaticalidad sobre determinadas cadenas de palabras. Los lingüistas aluden a estos juicios con el nombre de **intuiciones**. Por desgracia, las intuiciones no se nos presentan abiertamente y a menudo no son simples oposiciones entre lo que "no está bién"(*) y lo "perfecto".

La indeterminación de las intuiciones constituye una seria perturbación en los datos del estudioso, por lo que éste debe encontrar medios de superarla. Los lingüistas están acostumbrados a ella y tratan de circunscribirse tanto como pueden a los casos más nítidos. Por lo mismo, como el hablante nativo del castellano no suele tener ninguna concepción espontánea sobre las intuiciones, los lingüsitas deben aprender a intervenir por sí mismos en ellas. En cierto modo éstos son expertos en lenguaje, ya que tienen como nadie una cierta idea sobre lo que es y lo que es el castellano. En consecuencia, cabe muy bien la posibilidad de que algunos de los juicios sobre gramaticalidad que aquí se presenten aun cuando resulten aceptables para un cierto sector de lingüistas profesionales, parezcan oscuros e incluso quizá perversos para el público corriente.

Un detalle importante aquí es el siguiente: La sintaxis teórica en modo alguno afirma que la gente hable todo el timpo con oraciones plenamente gramaticales, pues es evidente que no ocurre así. Son muchos los obstáculos mentales y físicos que hay que superar en toda situación de habla. ¿Acaso esto derrumba el fundamento en que se basa el estudio de la sintaxis? De ninguna manera, puesto que lo importante es que, tras la abstracción de los detalles concretos del habla real, pueden obtenerse datos bastante fiables sobre lo que es y lo que no es castellano[2].

[2] Para utilizar un símil, los filósofos y semantistas se interesan mucho por las condiciones de verdad respecto del significados. Así, en condiciones normales se emplearía la oración *Llueve en Buenos Aires* si fuese efectivamente el caso de que lloviera en Buenos Aires y los hablantes nativos convinieran en que es así. Sin embargo, no se da semánticamente en ninguna lengua humana que sus hablantes digan (siempre) la verdad; al contrario, es evidente que no lo hacen. (Con ello no se está negando que exista una cierta utilidad comunicativa en decir la verdad y que la gente suele decirla efectivamente. Los grandes mentiorsos lo son por decir grandes mentiras y no por mentir en cada enunciado. Análogamente, la gran mayoría de los enunciados

El último comentario sobre las intuiciones es más bien una advertencia. Las intuiciones que tenemos se aplican a la estructura asignada justamente al ejemplo que se está considerando. Pero con imaginación y tiempo suficientes, es probable que la mayoría de las secuencias de palabras proscriptas aquí por agramaticales podrían a la postre encontrarse aceptables. Por ejemplo, todos los lingüistas le dirían que lo siguiente es agramatical:

(1.4)

*Reagan thinks bananas.

¿Por qué? Pues porque el verbo *think* toma por complemento algo de naturaleza oracional o clausal. Sin embargo aquí parece que hemos dado a *thinks* un sintagma nominal como objeto directo, como en el caso de *Reagan sells bananas.* Con esta estructura sintáctica, el ejemplo resulta agramatical porque no se ajusta a las reglas del inglés.

Ahora bien, la cadena *Reagan thinks bananas* es aceptable de hecho, como se desprende de:

(1.5)
What is Kissinger's favorite fruit?

—Reagan thinks bananas.

Ahora lo entendemos como un complemente clausal, tal como rezan las reglas, y se produce una interpretación a base de *Reagan thinks (that) bananas (are Kissinger's favorite fruit)*, con las palabras entre paréntesis omitidas. La confirmación de que el complemento de *think* debe ser una oración no hace sino demostrar que lo importante es investigar la estructura asignada a una cadena más que la cadena en sí misma.

1.2.5 Constituencia

Al referirnos a la estructura de una cadena de palabras se ha puesto de manifiesto que la tarea de la sintaxis no consiste tan sólo en caracterizar cadenas del castellano, sino también en asignarles una estructura adecuada. Determinar exactamente cuál es la estructura correcta en un caso determinado es tan complicado que no vamos a poder entrar en ello aquí. Por lo demás, en el marco de una presentación de distintas teorías sintácticas tampoco resulta decisivo hacerlo, pues en general hay un acuerdo bastante extendido sobre las estructuras que hay que asignar. Las diferencias de opinión surgen más bien sobre la información que contienen las estructuras y sobre cómo se las arregla la teoría para relacionar las estructuras presentes en la lengua con las ausentes.

Las partes de sintaxis que se componen de subpartes más pequeñas de sintaxis se denominan **constituyentes**. Por ejemplo, el sintagma nominal "algunos pescados malolientes" es un constituyente compuesto de un determinante, un adjetivo y un nombre. Otro constituyente se encuentra en el sintagma verbal (una frase que suele contener un verbo y algún otro material más) "molestaron

son gramaticales, sobre todo en los estilos cuidados).

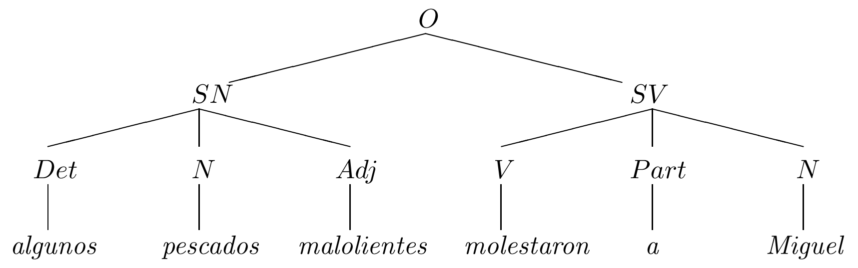
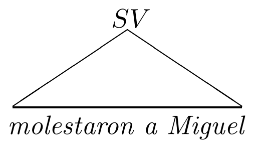


Figura 1.1: Estructura de constituyentes de la oración "Algunos pescados malolientes molestaron a Miguel".

a Miguel". Combinando estos dos constituyentes podemos obtener una oración, cuya estructura mínima es como se ve en la figura 1.1.

La oración (O) consta de un sintagma nominal (SN) y de un sintagma verbal (SV). Las reglas que afectan a los sintagmas nominales afectarán por igual, en condiciones ideales, tanto al SN *algunos pescados malolientes* como a *Miguel*. Por otro lado, no es de esperar que haya reglas que afecten a la secuencia *malolientes molestaron a*, por la sencilla razón de que no es un constituyente. En realidad, si hubiese una regla así, serviría de evidencia para sostener la hipótesis de que se trata efectivamente de uno, tal vez en contradicción con otras observaciones.

A veces conviene suprimir la estructura interna de un constituyente por no ser pertinente o interesante para lo que se está tratando. En tal caso se representa así:



1.2.6 Diagramas arbóreos

La estructura de la figura 1.1 se conoce por **diagrama arbóreo** o, más sucintamente, por un **árbol** que representa la estructura sintáctica de la oración o del sintagma en cuestión. Las propiedades que interesan a la lingüística son las siguientes:

- Una oración de la lengua es una cadena que lleva asignada una estructura cuya **raíz** se encuentra en la categoría O (es decir, O es el rótulo de categoría más alto). Puede haber otros casos de O dentro de la O inicial.
- El **nodo** del árbol llamado O **domina** todo lo que se encuentra en el árbol. Luego, O **domina inmediatamente** SN y SV. La relación de dominación reviste gran importancia, pues son muchas las reglas y operaciones sintácticas que se ejecutan en función de ella.
- El nodo O es el **padre** de SN y SV, y SN y SV son **hermanos**.
- Un **constituyente** es todo sector del árbol que tiene un solo padre.

Estos son algunos de los rasgos básicos de los árboles sintácticos.

1.2.7 Parentización rotulada

A veces conviene, o simplemente resulta más sugestivo, utilizar una representación diferente de la constituencia. Se trata de la **Parentización Rotulada**, que contiene la misma información que el árbol si bien representada en forma lineal. Así, la parentización rotulada del ejemplo de la figura 1.1 es como sigue:

(1.6)
$[_O[_{SN}[_{Det}algunos][_N pescados][_{Adj}malolientes]]$

$[_{SV}[_V molestaron][_{Part}a][_N Miguel]]]$
Es indudable que si no se está acostumbrado a estas presentaciones se hacen difíciles de seguir a primera vista. Sin embargo son muy comunes en la descripción sintáctica y merecen ser mencionadas.

1.2.8 Reglas de estructura sintagmática

Los árboles representan la estructura de los sintagmas y oraciones de la lengua. La siguiente pregunta es acerca de dónde proceden los árboles. Es decir, ¿Cómo nos dice la teoría sintáctica qué estructuras están bien formadas y cuáles no? La respuesta más elemental es que se confecciona un conjunto de reglas que **generan** árboles y que nuestro conocimiento de la sintaxis consiste justamente en saber cuáles son estas reglas. Por lo que hemos visto hasta aquí, las siguientes **reglas de estructura sintagmática** (reglas ES) generarán el árbol de la figura 1.1:

	(a)	O	$\rightarrow SN\ SV$
(1.7)	(b)	SN	$\rightarrow Det\ N\ Adj$
	(c)	SV	$\rightarrow V\ Part\ SN$

Técnicamente, el vector significa "se rescribe como" (esto es, si se tiene una O, puede "rescribirse como" SN y SV). Desde un punto de vista más lingüístico, podemos interpretarlo a base de "extenderse como" (es decir, "dominar en el árbol").

En general se suelen incluir dos variaciones. La primera se refiere a la recursión y la segunda, a la opcionalidad. De este modo, para verbos tales como *creer* tienen sentido reglas tales como

(1.8)

$SV \rightarrow V\ O$
En efecto, ahora la regla (1.8) puede "nutrir" (1.7) (a) y generar Os dentro de Os, nuestro sistema llegará a generar tantos objetos como queramos. Se trata, por cierto, de un elemento decisivo para captar la naturaleza de la flexibilidad y la creatividad del lenguaje.

La noción de opcionalidad llama un poco menos la atención. Viendo de nuevo (1.7) (b) vemos que un SN no necesita necesariamente de la existencia del determinante y del adjetivo. También hay verbos intransitivos, por lo que al SN dentro de SV lo podemos considerar opcional. Esto se expresa como sigue

$$(1.9) \quad \begin{array}{ll} O & \to SN\,SV \\ SN & \to (Det)\,N\,(Adj) \\ SV & \to V\,(Part\ SN) \end{array}$$

Todo ello proporciona una visión panorámica sobre el modo de producir árboles. Pero como las oraciones son árboles con palabras en las hojas, el próximo paso consistirá en mostrar cómo se incorporan las palabras.

1.2.9 El lexicón

Parte de nuestro saber lingüístico comprende un gran número de palabras, que constituyen nuestro vocabulario y forman lo que los lingüistas llaman el **lexicón**. En general los elementos del lexicón se corresponden con lo que entendemos por palabras, auqnue hay teorías sintácticas que tienen concepciones ligeramente distintas sobre esos elementos o "datos léxicos". De ahí que no siempre resulte del todo razonable imaginar el lexicón como un mero acopio de palabras. En particular, el lexicón de una Gramática Generativa puede contener una lista de diversos afijos, como el sufijo verbal -*aba* del castellano (por el que se distingue *cantaba* de *canta*.)

1.2.10 Inserción léxica

Cuando se insertan las entradas léxicas de la categoría adecuada en la parte terminal de un árbol se obtiene una oración. Las entradas léxicas de tipo *pescado*, *alto* o *rápidamente* se denominan **símbolos terminales** porque con ellos, en cierto modo, termina el árbol. Las categorías como N, V y Adj se llaman **preterminales**. A continuación se presenta la inserción léxica mediante reglas ES, a las que añadimos una nueva notación a base de llaves para indicar distintos puntos de elección. Por ello (1.10) establece que V puede dominar inmediatamente *estornudar* o *dormir* o *cocinar*, etcétera. Por lo demás cambiamos la abreviatura Adj por A, más habitual, para "Adjetivo":

$$(1.10) \quad \begin{array}{ll} V & \to \{\text{estornudar, dormir, cocinar, decir, \ldots}\} \\ S & \to \{\text{pescado, hombre, desesperación, bañera, \ldots}\} \\ A & \to \{\text{maloliente, alto, confiado, falso, \ldots}\} \end{array}$$

Nada impide que lo que parece una misma palabra figure en distintas categorías —como en el caso de *paro* que es nombre y verbo—. En estos casos es mejor considerar que se trata de dos elementos léxicos diferentes, aunque relacionados entre sí, pues casualmente se pronuncian del mismo modo.

1.2.11 Subcategorización

Las entradas léxicas allegan grandes cantidades de información sobre sus respectivos datos. Piénsese tan sólo la entrada media de un buen diccionario convencional. Así, por ejemplo, la entrada del dato léxico "dar", contendrá información de que pertenece a la categoría sintáctica del verbo, que se pronuncia de una determinada manera, que significa tal o cual cosa, y así sucesivamente. Uno tipo de información muy importante que contienen los datos léxicos es el que los lingüistas llaman **subcategorización**. La ilustración más simple se encuentra en la diferencia que hay entre un verbo transitivo y un verbo intransitivo, por la cual el verbo transitivo debe llevar un objeto para ser gramatical, en tanto que un verbo intransitivo no debe llevarlo, como se desprende de los siguientes ejemplos:

(1.11)
(a) César murió.
(b) *César engendró.
(c) *César murió cuatro hijos.
(d) César engendró cuatro hijos.

Si ampliamos las reglas que hemos dado con los elementos léxicos adecuados podremos generar estos cuatro ejemplos. Lo único que necesitamos es dividir la clase de los verbos en **subcategorías**, a saber de los intransitivos, de los transitivos, etcétera . Con ello debemos añadir en la entrada léxica de "morir", la restricción de que **sólo** debe insertarse en una estructura sintáctica sin objetos detrás de V, en tanto que deberemos estipular lo contrario para "engendrar." (También los verbos pueden pertenecer a más de una subcategoría, lo que sucede, por cierto, a menudo —v. gr., "comer", es un conocido ejemplo de verbo que puede ser transitivo e intransitivo).

A continuación se presenta una muestra de entradas léxicas de verbos seguidos de algunos sintagmas verbales ilustrativos:

	morir, V, —	
	estornudar, V, —	
	comer, V, —	
	comer, V, — SN	*comer papas fritas*
(1.12)	engendrar, V, — SN	*engendrar cuatro hijos*
	devorar, V, — SN	*devorar la albóndiga*
	dar, V, — SN SP	*dar una galleta a Juan*
	creer, V, — O	*creer que Max duerme*
	decir, V, — SN O	*decir a Susy que todo cambia*

La sexta línea estipula, por ejemplo que "devorar" es un verbo y que debe aparecer en un contexto inmediatamente antepuesto a un SN. Decimos entonces que **subcategoriza** el SN y que la última parte de la entrada (después de la segunda coma) es el **marco de subcategorización**. En la séptima línea se establece que "dar", subcategoriza a un sintagma nominal (SN) y un sintagma preposicional (SP) y en ese orden.

Capítulo 2

El formalismo de las Gramáticas Léxico-Funcionales

La teoría de las grámaticas léxico funcionales (LFG de aquí en adelante) ha sido, desde su introducción por Kaplan y Bresnan [21], aplicada en la descripción de una amplio rango de fenómenos lingüísticos. En esta sección vamos a hablar un poco acerca de la historia de la LFG y en las secciones subsiguientes se la explicará más en detalle. Las bases del formalismo son bastante simples: la teoría asigna dos niveles de representación sintáctica a cada oración, la estructura de constituyentes y la estructura funcional. La estructura-c es un árbol de estructura de frase que sirve como base para la interpretación fonológica mientras que la estructura-f es una matriz jerárquica de atributos-valores que representan relaciones gramaticales subyacentes. La estructura-c es asignada por reglas de una gramáctica de estructura de frase libre de contexto (ver capítulo siguiente sobre gramáticas libres de contexto). Se agregan anotaciones funcionales sobre dichas reglas para que al ser instanciadas provean ciertas restricciones sobre las estructuras-f posibles, y la estructura más pequeña que satisfaga esas restricciones será la gramáticamente apropiada [22].

Esta concepción formal evolucionó desde mediados de la década del '70 desde unos tempranos trabajos en lingüística teórica y computacional. Las *Agumented Transition Networks* de Woods [43] demostraron que un mapeo directo entre las estructuras superficiales y subyacentes era suficiente para codificar la discrepancia entre la forma externa de las expresiones y sus relaciones internas predicado-argumento. Las gramáticas ATN se alinean con la gramática transformacional al usar el mismo tipo de estructuras matemáticas, árboles de estructura de frase, tanto como en la representación gramatical superficial y profunda. Ronald Kaplan en 1975 [20] se dió cuenta que la fuerte motivación transformacional para esta similitud de representación no existía en la teoría ATN. Las entradas y salidas de las transformaciones tienen que ser del mismo tipo formal si las reglas están por ser retroalimentadas en una secuencia de derivaciones, pero una aproximación no-derivativa no impone tal requerimiento. Luego, mientras que estructuras jerárquicas y de árbol ordenado son apropiadas para representar la secuencia de de palabras y oraciones superficiales, no hay nada que haga particularmente conveniente expresar relaciones gramaticales más abstractas entre las funciones gramaticales y sus atributos. Si bien el hecho que *DeepBlue* es el sujeto en *DeepBlue juega ajedrez* puede ser formalmente representado en un árbol en el cual *DeepBlue* es el SN directamente debajo del nodo S, no hay ninguna ventaja explicativa en usar una forma tan complicada de codificar una intuición tan sencilla. Kaplan [20] propuso matrices jerárquicas de pares atributo-valor, ahora conocidas como estructuras-f, como un modelo más natural de representar las relaciones gramaticales subyacentes.

Las operaciones de manejo de registros permitían acceso explícito a etiquetas tales como Sujeto u Objeto. Originalmente se las usaba para manipular información temporal que se acumulaba durante el proceso de análisis de una oración y que era reorganizada al final del proceso para formar una tradicional estructura profunda transformacional. Kaplan [20] no vió necesidad de tal reorganización, ya que los registros acumuladores tenían ya toda la información gramatical significativa. Pero este cambio en la importancia de los registros desde repositorios de información intermedia necesaria a ser el destino del más importante análisis lingüístico tuvo mucho más amplias consecuencias. La naturaleza exacta de las operaciones de disposición y acceso de los registros resultó en un tema de gran importancia teórica, y estudios teóricos eran también requeridos para las configuraciones particulares de los contenidos de los registros que luego la gramática asociaba con las oraciones individuales. El formalismo LFG emergió de un estudio cuidadoso de preguntas de este tipo. La información acumulada en los registros fue formalizada como funciones monádicas definidas sobre un conjunto de relaciones gramaticales y nombres de atributos (SUJ, OBJ, $CASO$), y las operaciones computacionales de las ATNs para manipular estas funciones evolucionaron en las especificaciones ecuacionales de las descripciones funcionales de las LFGs.

Esta maquinaria formal ha servido como base para importantes investigaciones sobre las propiedades en común de todos los lenguajes humanos y sobre las particularidades de determinados lenguajes. Las investigaciones tempranas establecieron, por ejemplo, el carácter universal de funciones gramaticales tales como sujeto y objeto, principios generales de control y concordancia, y los mecanismos básicos para expresar e integrar información sintáctica y léxica [21]. Estos estudios y resultados más recientes han ofrecido un fuerte apoyo para la organización de la teoría, pero también han descubierto problemas particularmente difíciles de resolver en ella tal como se la había formulado originalmente. Es por ello, entonces, que ciertas revisiones y extensiones a la LFG están actualmente en consideración, tratando con dependencias a gran distancia, coordinación, orden de las palabras e interpretación semántica y pragmática. Algunas de estas propuestas [33] pueden parecer cambios un poco radicales pero la arquitectura subyacente presentada en [21] se mantiene aun bastante compatible.

2.1 Convenciones de notación en LFG

En esta sección y las subsiguientes vamos a hacer una breve introducción al formalismo [42], de modo que queden en claro los términos *Estructuras de Constituyentes Anotadas* (estrucuras-c) y de *Estructuras Funcionales* (estructuras-f), así como también su proceso de generación. La razón por la que se usa la palabra *anotadas* es por que se emplean árboles rotulados con expresiones o *anotaciones*. Estas anotaciones son interpretables, una vez se hubo llenado el árbol con ellas. Este proceso es denominado de *instanciación* y se describe en la sección 2.2. El resultado de la instaciación es un conjunto de expresiones conocido como *Descripción Funcional*, el cual determina completamente como debemos construir la correspondiente estructura-f (sección 2.4). En la sección 2.5 se verá la naturaleza de las estructuras-f y los procedimientos para crearlas a partir de las descripciones funcionales. Sabiendo ya construir estructuras-f se pasará a cuestiones más lingüísticas concernientes con el rol de las estructuras-f en la predicción de la buena formación de las oraciones del lenguaje natural (sección 2.6). En la sección 2.7 se tocan algunos temas formales referidos a ecuaciones funcionales especiales.

2.1.1 La forma sintáctica de las reglas en LFG

En su forma exterior las reglas en LFG recuerdan las reglas libres de contexto que son el componente base de la gramática transformacional en la Teoría Estándar. Las reglas de una Gramática Léxico-Funcional, sin embargo, contienen expresiones conocidas como *Esquema Funcional*, las cuales están asociadas con los símbolos que aparecen al lado derecho de la flecha →. La figura 2.1 muestra el formato usual de escritura de reglas en LFG.

Las siguientes reglas se pueden combinar con un lexicón para generar un pequeño fragmento del castellano.

$$
(2.1)\quad
\begin{array}{lllcc}
\text{(a)} & O & \rightarrow & \underset{(\uparrow SUJ)=\downarrow}{SN} & \underset{\uparrow=\downarrow}{SV} \\[2ex]
\text{(b)} & SV & \rightarrow & \underset{\uparrow=\downarrow}{V} & \underset{(\uparrow OBJ)=\downarrow}{SN} \\[2ex]
\text{(c)} & SN & \rightarrow & \underset{\uparrow=\downarrow}{DET} & \underset{\uparrow=\downarrow}{N}
\end{array}
$$

Las expresiones $(\uparrow SUB) =\downarrow$, $\uparrow=\downarrow$ y $(\uparrow OBJ) =\downarrow$ son todas esquemas funcionales. Se discutirá en la sección 2.2 como se interpretan estas reglas para crear estructuras-c anotadas.

2.1.2 El formato de los ítems léxicos en LFG

Como en las reglas recién descriptas, a los ítems léxicos se los suplementa también con esquemas funcionales en LFG. En muchos trabajos relacionados con LFG encontramos entradas léxicas que contienen al menos tres cosas: una representación de la forma del ítem, la categoría sintáctica a la que el ítem pertenece, y una lista de esquemas funcionales. Este se escribe usualmente como se muestra en la figura 2.2.

Los siguientes tres ítems léxicos pueden ser combinados con las reglas (2.1) para generar un par de oraciones castellanas.

$$
(2.2)\quad
\begin{array}{lll}
\text{(a) DeepBlue} & N & (\uparrow PRED) = \text{`DEEPBLUE'} \\
 & & (\uparrow NUM) = SING \\
 & & (\uparrow PERS) = 3 \\[2ex]
\text{(b) Juega} & V & (\uparrow PRED) = \text{`JUGAR} < (\uparrow SUJ)(\uparrow OBJ) > \text{'} \\
 & & (\uparrow SUJ\ \ NUM) = SING \\
 & & (\uparrow SUJ\ \ PERS) = 3 \\[2ex]
\text{(c) Ajedrez} & N & (\uparrow PRED) = \text{`AJEDREZ'} \\
 & & (\uparrow NUM) = SING \\
 & & (\uparrow PERS) = 3
\end{array}
$$

2.2 Creando estructuras de constituyentes anotadas

En esta sección construiremos la estructura anotada de constituyentes para la oración en (2.3).

(2.3)

DeepBlue juega ajedrez.

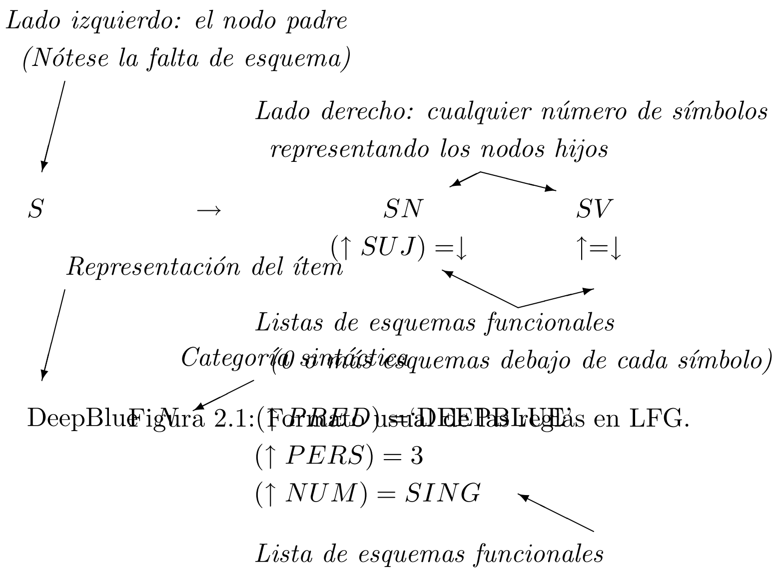


Figura 2.1: Formato usual de las reglas en LFG.

Figura 2.2: Formato usual de los ítems léxicos en LFG.

La representación de la estructura-c es la fuente de dos tipos de información. Primero, indica la estructura jerárquica de los constituyentes frasales en una cadena de una manera familiar a la mayoría de los lingüistas. Más importante aun, las llamadas **anotaciones funcionales** (esquemas funcionales transferidos dentro de los árboles) pueden ser interpretados para derivar información acerca de la estructura funcional.

Como las reglas de estructura de frase libres de contexto y los árboles de estructura de frase son generalmente objetos familiares, la creación de una estructura-c será una tarea más bien sencilla. El único trabajo adicional será la inserción del esquema funcional; sin embargo, esto es bastante poco complicado.

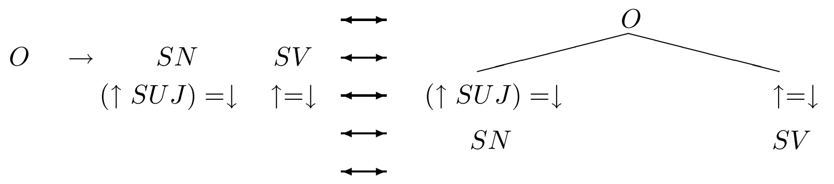


Figura 2.3: Relación entre reglas y anotaciones en un árbol.

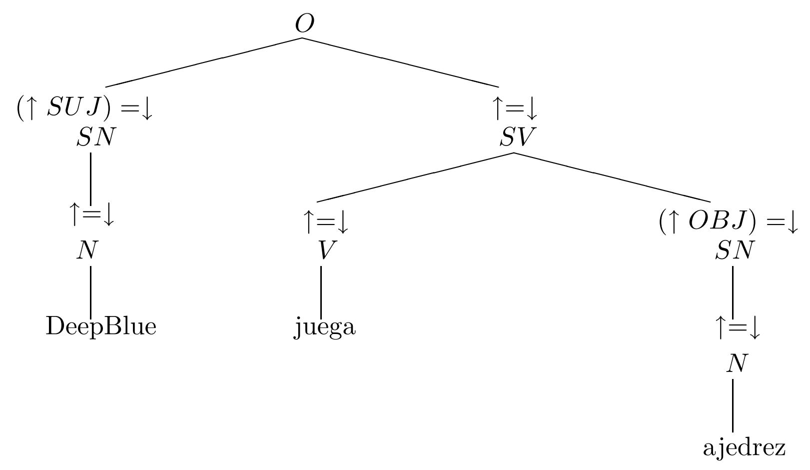


Figura 2.4: Árbol de la oración *DeepBlue juega ajedrez.*

Primero hay que considerar las reglas sintácticas. Cuando una regla se aplica, una parte del árbol se construye y las anotaciones prescriptas por las reglas son escritas encima de los nodos apropiados en la forma esquematizada en la figura 2.3.

Por lo tanto, las reglas en (2.1) (a)-(c) predicen que (2.3) dominará un árbol como el de la figura 2.4.

Esto representa, sin embargo, sólo una etapa en la costrucción del árbol anotado para (2.3); la estructura-c no estará completa sino hasta después de introducir las anotaciones especificadas en las entradas léxicas de *DeepBlue, juega* y *ajedrez.* De este modo, por cada elemento léxico en el árbol, consultamos la entrada léxica correspondiente y copiamos todo el esquema funcional encontrado en dicha entrada al árbol, encima de la palabra apropiada. La figura 2.5 da la estructura anotada de constituyentes final para (2.3).

Ahora que los esquemas funcionales ha sido transferidos dentro del árbol, ya están listos para ser interpretados. Como veremos en la sección siguiente, es la ubicación dentro del árbol lo que eventualmente da a los esquemas algún significado.

2.3 Instanciación

A partir de la estructura-c anotada discutida en las secciones previas uno puede construir estructuras-f, el otro nivel de representación sintáctica empleado en las Gramáticas Léxico Funcionales. Las flechas $\uparrow$ y $\downarrow$ empleadas en el esquema asumen un valor referencial ahora que éste ha sido situado en el árbol: ciertamente, se puede ver que estos caracteres en flecha han sido elegidos por su valor simbólico, ya que estas *apuntan,* por ponerlo de algún modo, a cosas arriba y abajo de la posición del esquema en el árbol. Determinar el referente de unas $\uparrow$ y $\downarrow$ y almacenar tal información en el esquema es lo que se conoce como **instanciación.** La instanciación transforma el esquema en **ecuaciones funcionales,** expresiones especificadas de manera completa en un lenguaje formal usado para hablar de estructuras-f. En la subsección 2.3.1 se verá la relación entre nodos en el árbol de la estructura-c y las estructuras-f. Con este conocimiento en la mano, podemos continuar con el problema de encontrar los referentes de las flechas $\uparrow$ y $\downarrow$.

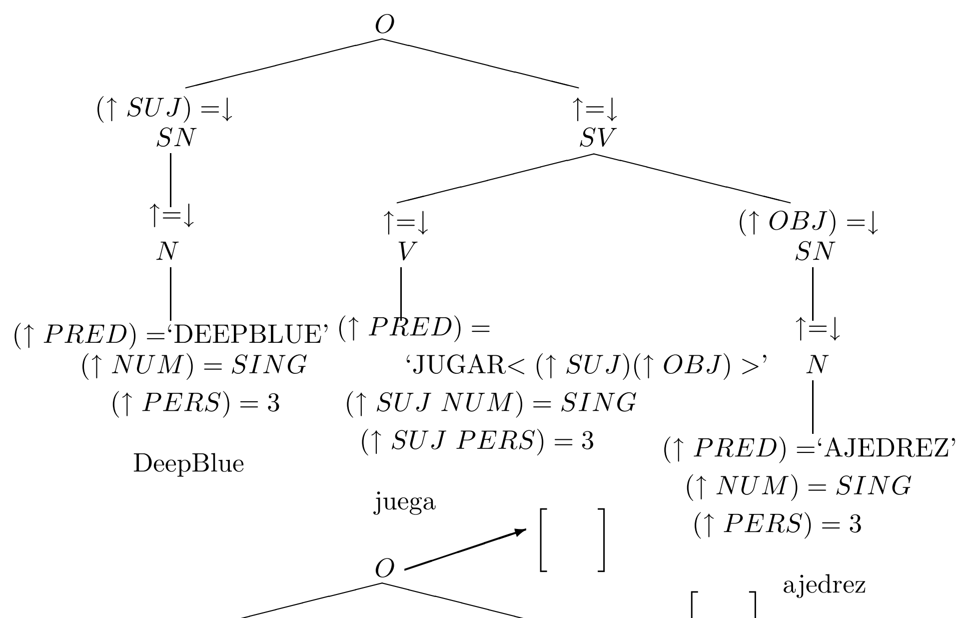


Figura 2.5: Árbol de la oración *DeepBlue juega ajedrez* con entradas léxicas.

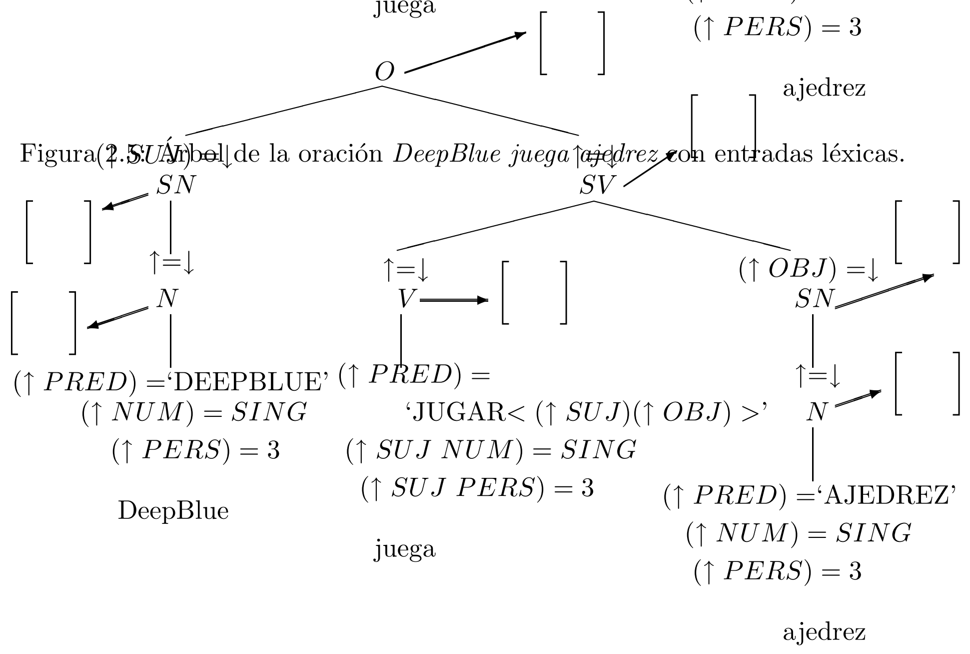


Figura 2.6: Relación entre los nodos en un árbol y las estructuras-f.

En esta sección se estudiarán las estructuras-f sin preocuparnos por el momento de sus contenidos. En otras palabras vamos a pensar en términos de *alguna estructura-f potencial*. Gráficamente las estructuras-f son representadas en la literatura como material encerrado en grandes corchetes, y por ahora usaremos pares de corchetes vacíos en las figuras para representar nuestras estructuras-f abstractas.

2.3.1 La relación entre nodos de la estructura-c y las estructuras-f

Una suposición de la LFG es la existencia de alguna estructura-f asociada con cada nodo en el árbol de constituyentes. La figura 2.6 esquematiza como debe conceptualizarse esta relación.

Para facilitar hablar acerca de estructuras funcionales, asociamos nombres arbitrarios o variables a cada estructura-f. De modo de prever cualquier posible confusión, hay que enfatizar que la elección de la variable para una estructura-f dada es *completamente arbitraria*. Hay una cierta tradición de utilizar variables que consisten de la letra f seguida de un número, por ejemplo f_{123}, diferentes números significando diferentes variables. Gráficamente se escribe la variable identificadora de cada estructuras-f sobre su esquina superior izquierda. Más aun, para expresar en una manera compacta la asociación de una estructura-f con su nodo en el árbol de constituyentes, podemos colocar el número de la variable de la estructura-f al lado del nodo que le corresponde. Este proceso se suele llamar **coindexar** al nodo. La conceptualización de la figura 2.6 puede ser elaborada como en la figura 2.7.

2.3.2 Encontrando los referentes de ↑ y ↓

Ahora podemos comenzar el proceso de instanciación de los esquemas en el árbol de forma tal de convertirlos en ecuaciones funcionales totalmente especificadas. El primer paso aquí es determinar a qué se refieren las flechas ↑ y ↓ en el árbol anotado. De hecho se refieren a estructuras-f, y todo lo que hace falta, entonces, para convertir estos esquemas en ecuaciones es insertar las variables que nombran las estructuras-f apropiadas en lugar de las fle chas. De este modo las flechas ↑ y ↓ son símbolos que pueden ser reemplazados por *variables* de tipo estructura-f, a las cuales se les suele llamar **metavariables** en la literatura.

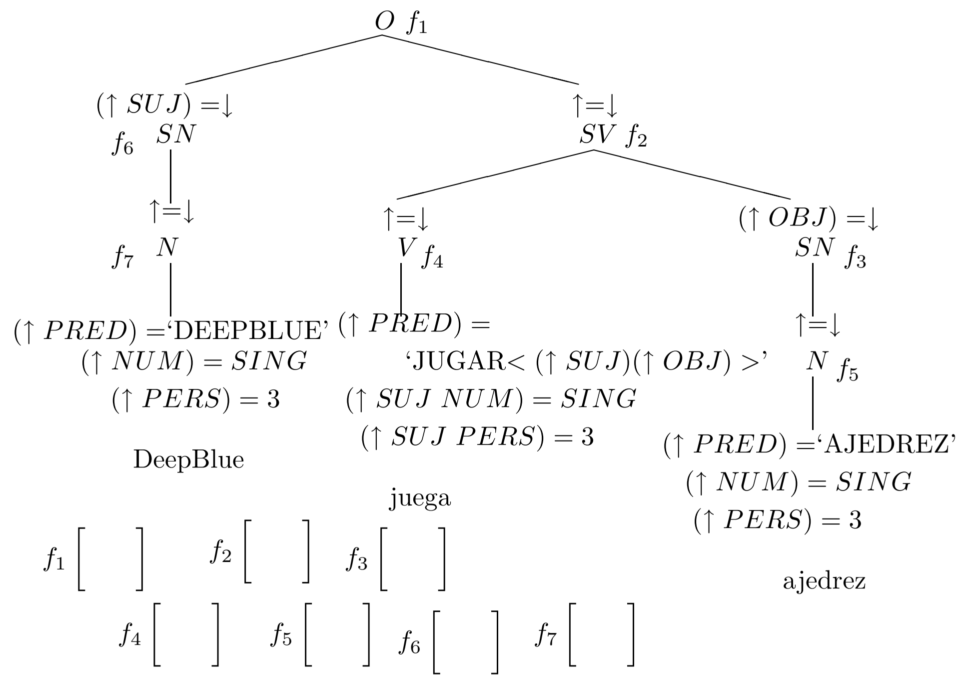


Figura 2.7: Proveyendo nombres para estructuras-f y coindexando los nodos del árbol.

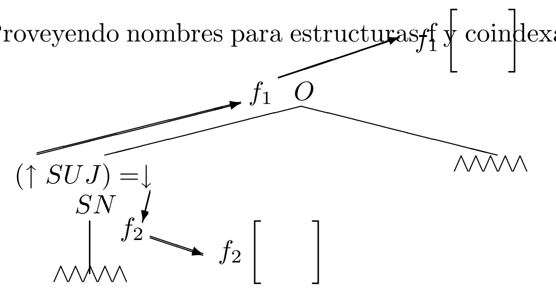


Figura 2.8: Determinando referentes para las metavariables Self y Madre.

Encontrar los referentes de las flechas $\uparrow$ y $\downarrow$ es un problema sencillo. La $\downarrow$ es conocida como la metavariable **ego** o **self**: se refiere a la estructura-f asociada con el nodo encima del cual aparece el esquema conteniendo el $\downarrow$. La otra metavariable, la $\uparrow$, es llamada la metavariable **madre**: se refiere a la estructura-f asociada con el padre del nodo encima del cual aparece el esquema conteniendo la $\uparrow$. Este conjunto de relaciones se esquematiza como en la figura 2.8, donde a las flechas $\uparrow$ y $\downarrow$ se las continua de forma de mostrar sus respectivos referentes.

Asi completamos el proceso de intanciación, reemplazando todas las metavariables con los nombre de las estructuras-f a los que se refieren, de una forma esquematizada en la figura 2.9, que es una modificación de la figura 2.8.

La figura 2.10 muestra el árbol terminado para el ejemplo (2.3).

2.3.3 Consolidación

Debería notarse que las secuencias de figuras siguientes con sus estructuras-f vacías, punteros y cosas por el estilo intentaban ser simplemente ayudas para conceptualizar como funciona el proceso de instanciación y lo que este representa. En la práctica no es necesario dibujar figuras complejas: el algoritmo esencial descripto más arriba puede ser resumido como sigue. Habiendo construido la estructura-c anotada ((a) en la figura 2.11 muestra parte de esta estructura), dar a cada nodo en la estructura-c una variable distinta, que luego dará nombre a la estructura-f de dicho nodo (ver (b) de la figura 2.11). La elección de las variables es completamente arbitraria, salvo la restricción de no asignar a dos nodos la misma variable. Una vez hecho ésto, considere cada esquema funcional, y reemplace todas las instancias de la metavariable self con la variable asociada con el nodo encima del cual aparece el esquema. También reemplace todas las instancias de la metavariable madre con la variable aplicada al padre del nodo encima del cual aparece el esquema (ver (c) de la figura 2.11).

2.4 La descripción funcional

El conjunto de todas las ecuaciones funcionales instanciadas en el árbol se denomina la **descripción funcional**. La descripción funcional para la figura 2.10 se verá más adelante. Es a partir de las descripciones funcionales que se construyen las estructuras-f, y para el resto del proceso constructivo el árbol ya no es utilizado. Este es un punto oportuno para hacer algunas advertencias respecto del contenido de la estructura-f. La gente cuando se acerca a LFG es muy común que espere que cierta información categórica sea transferida desde el árbol a la estructura-f (por ejemplo, que *verbo* sea mencionado de algún modo en la estructura-f). Hay que hacer hincapié en el hecho de que sólo la información presente en las ecuaciones de las descripciones funcionales será incorporada eventualmente en la estructura-f.

(2.4)

(a) $(f_1\ SUJ) = f_2$
(b) $f_2 = f_3$
(c) $(f_3\ PRED) =$'DEEPBLUE'
(d) $(f_3\ NUM) = SING$
(e) $(f_3\ PERS) = 3$
(f) $f_1 = f_4$
(g) $f_4 = f_5$
(h) $(f_5\ PRED) =$'JUGAR $< (f_5\ SUJ)(f_5\ OBJ) >$'
(i) $(f_5\ SUJ\ NUM) = SING$
(j) $(f_5\ SUJ\ PERS) = 3$
(k) $(f_4\ OBJ) = f_6$
(l) $f_6 = f_7$
(m) $(f_7\ PRED) =$'AJEDREZ'
(n) $(f_7\ NUM) = SING$
(o) $(f_7\ PERS) = 3$

2.5 Estructuras-f: su naturaleza y construcción

La subsección 2.5.1 discute algunas características de las estructuras-f. Pasando luego a la subsección 2.5.2 se abordará la pregunta de qué significan las ecuaciones funcionales con respecto a las estructuras-f que describen. La subsección 2.5.3 describe el proceso de construir una estructura-f, usando la oración (2.3) como un ejemplo para clarificar muchos de los puntos más confusos de este procedimiento.

2.5.1 Características de las estructuras-f

Siguen ahora algunas observaciones básicas acerca de las estructuras-f.

La forma de las estructuras-f

Gráficamente una estructura-f toma la forma mostrada en (2.5).

(2.5)

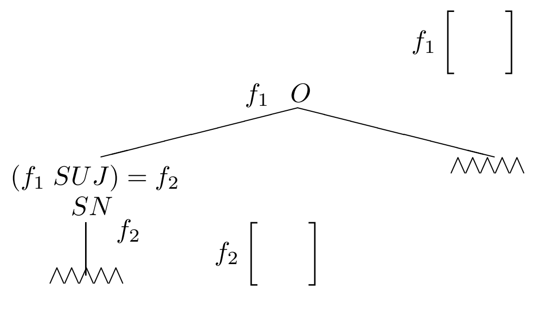


Figura 2.9: Completando la instaciación mediante coindexación.

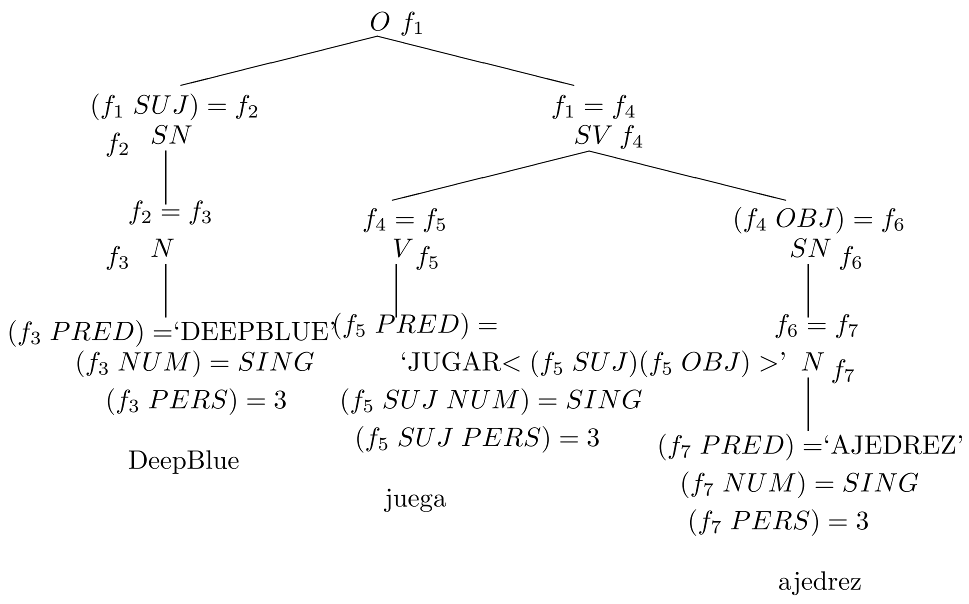


Figura 2.10: Árbol completo para la oración *DeepBlue juega ajedrez.*

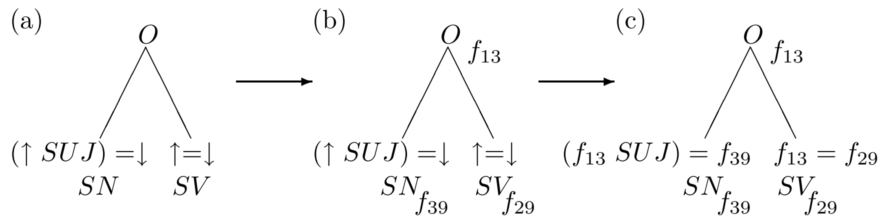


Figura 2.11: Algoritmo de instaciación conciso.

$$\begin{bmatrix} A & f_m \begin{bmatrix} B & C \\ D & E \end{bmatrix} \\ F & G \\ H & I \end{bmatrix}$$

Cada estructura-f consiste en dos columnas de entradas encerradas en grandes corchetes. La columna de la izquierda contiene lo que se conoce como **atributos** y la otra, **valores**. Atributos y valores se encuentran de a pares, y los miembros de un par se escriben en la misma línea horizontal. Los atributos son siempre símbolos sencillos, como A, SUJ, $PRED$ o cualquier otra cosa que un lingüista desee usar. Los valores, sin embargo, pueden ser símbolos sencillos, estructuras-f subordinadas, o formas semánticas. Las formas semánticas son reconocibles por el hecho de que están siempre rodeadas de comillas simples; serán discutidas más adelante.

Fuera del corchete izquierdo de una estructura-f es posible escribir de forma opcional el nombre de la estructura-f, tal como vimos en (2.5).

Se avisa al lector respecto de que no hay absolutamente ningún significado en el orden en que ocurren las líneas de una estructura-f. De hecho, (2.6) (a) y (2.6) (b) son dos representaciones de la *misma* estructura-f.

(2.6) (a) $\begin{bmatrix} A & B \\ C & D \\ E & \begin{bmatrix} F & G \\ H & I \end{bmatrix} \end{bmatrix}$ (b) $\begin{bmatrix} C & D \\ E & \begin{bmatrix} H & I \\ F & G \end{bmatrix} \\ A & B \end{bmatrix}$

La condición de unicidad sobre las estructuras-f

Las estructuras-f deben satisfacer una condición de unicidad: si hay un valor, digamos K, asociado a un atributo J en f_p, y otro valor distinto L está también asociado a J en f_p, entonces la estructura-f estará mal formada.

(2.7) $^*f_p \begin{bmatrix} \ldots & \ldots \\ J & K \\ J & L \\ \ldots & \ldots \end{bmatrix}$

2.5.2 El significado de las ecuaciones funcionales

Como ya se ha establecido, las ecuaciones funcionales son expresiones con sentido en un lenguaje formal: tienen información acerca de las estructuras-f. Habiendo ganado un conocimiento rudimentario de la forma de las estructuras-f, estamos ahora en posición de discutir que expresan las ecuaciones funcionales, esto es, de proveer una semántica a este lenguaje formal. La figura 2.12 nos da un esquema para interpretar las ecuaciones funcionales.

Tomemos un ejemplo concreto, digamos la ecuación en (2.8)

(2.8)

$(f_n\ F) = G$

$(f_p\ ATR) = VAL$

En la estructura-f f_p hay una línea donde,
el atributo es ATR,
y el valor es VAL,

Figura 2.12: Significado de las ecuaciones funcionales.

De acuerdo con esta ecuación, la estructura-f f_n contiene una línea donde F es un atributo y G es su valor. Digamos que la estructura-f delineada en (2.5) sea f_n. Comparemos ahora la ecuación de más arriba con f_n (i.e., (2.5)). Vemos que realmente hay una línea en f_n (la segunda desde el final) donde F es un atributo y G su valor. Por lo tanto, para nuestra f_n, la ecuación (2.8) es verdadera. Consideremos ahora otra ecuación, asumiendo que nuestra f_n es todavía (2.5).

(2.9)

$(f_n\ Q) = R$

Como no hay ninguna línea en f_n donde el atributo sea Q y el valor sea R, (2.9) es falsa. En (2.10) enumeramos cinco ecuaciones que son verdaderas para (2.5). Nótese que la primera de estas dice que la estructura-f f_m es el valor de A en f_n y que la segunda dice algo acerca de los contenidos de f_m.

(2.10)

$(f_n\ A) = f_m$
$(f_m\ B) = C$
$(f_m\ D) = E$
$(f_n\ F) = G$
$(f_n\ H) = I$

2.5.3 La construcción de estructuras-f

Si uno comprendiera qué es lo que las ecuaciones de una descripción funcional están diciendo acerca de una estructura-f dada, uno podría ciertamente dibujar dicha estructura-f: el propósito es construir una estructura-f de forma tal que todas las ecuaciones funcionales contenidas en la descripción funcional sean ciertas. Para comenzar, tomemos (2.11) como un ejemplo de una descripción funcional.

(2.11)

$(f_r\ A) = B$
$(f_r\ C) = D$

Tomemos la primera de estas dos ecuaciones y consideremos qué haría falta para hacerla verdadera. Tendría que haber una estructura-f llamada f_r, y ésta debería contener una línea donde A sea el atributo y B su valor. Ahora si conocemos que la estructura-f tiene estas características, podemos adelantarnos y comenzar a construirla, insertando todas las particularidades que acabamos de determinar que debería tener.

(2.12)

$$f_r \left[A \quad B \right]$$

Podemos ahora movernos a la siguiente ecuación, la cual requiere lo siguiente para ser cierta: f_r debe tener una línea donde C sea el atributo y D su valor. Con este conocimiento podemos modificar la representación de (2.12) para que refleje la nueva información.

(2.13)

$$f_r \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Minimalidad

En (2.13) tenemos una estructura-f llamada f_r la cual hace ambas ecuaciones en (2.11) verdaderas. Sin embargo, es importante notar que lo que se busca es la estructura-f **minimal.** Debemos mejor discutir minimalidad en referencia a un ejemplo. Consideremos la descripción funcional en (2.11) en relación a las dos estructuras-f en (2.14).

(2.14) (a) $f_r \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ (b) $f_r \begin{bmatrix} A & B \\ C & D \\ E & F \end{bmatrix}$

la estructura-f en (2.14)(a) es la misma que ya hemos visto, y la que ya hemos observado que hace ambas ecuaciones en (2.11) verdaderas. Fijémonos también que ambas ecuaciones en (2.11) son verdaderas para la estructura-f de (2.14)(b). Ahora observemos que si sacamos cualquiera de las dos líneas que componen (2.14)(a), una u otra de las ecuaciones en (2.11) deja de ser cierta. Cuando todas las ecuaciones funcionales en una descripción funcional son ciertas en una dada estructura-f, y el sacar una línea cualquiera de la estructura-f puede invalidar alguna ecuación en la descripción funcional, entonces y sólo entonces dicha estructura-f será considerada la estructura-f *minimal* para la descripción funcional en cuestión. Considérese ahora (2.14)(b). Como hemos notado, ambas ecuaciones en (2.11) son verdaderas para (2.14)(b) como se muestra arriba. Sin embargo, podemos sacar una línea de (2.14)(b) (específicamente la última línea), y ambas ecuaciones seguirán siendo ciertas para la estructura-f resultante. Por lo tanto, (2.14)(b) *no es* una estructura-f minimal para la descripción funcional en (2.11).

Construcción de un ejemplo de estructura-f

En este punto vamos a proceder a construir una estructura-f para (2.3). Al hacer ésto vamos a recalcar alguno de los aspectos más dificultosos en la construcción de estructuras-f partiendo de las especificaciones de las ecuaciones funcionales.

Ecuaciones simples. Hay un gran grupo de ecuaciones funcionales en (2.4) que pueden ser manejadas fácilmente, dado el análisis anterior, esto es, lo dicho relacionado con los ejemplos (2.11) a (2.13).

$$(2.15)\quad \begin{array}{ll} \text{(c)} & (f_3\ PRED) = \text{`DEEPBLUE'} \\ \text{(d)} & (f_3\ NUM) = SING \\ \text{(e)} & (f_3\ PERS) = 3 \\ \text{(h)} & (f_5\ PRED) = \text{`JUGAR} < (f_5\ SUJ)(f_5\ OBJ) > \text{'} \\ \text{(m)} & (f_7\ PRED) = \text{`AJEDREZ'} \\ \text{(n)} & (f_7\ NUM) = SING \\ \text{(o)} & (f_7\ PERS) = 3 \end{array}$$

Hay tres nombres de estructuras-f usados en las ecuaciones de (2.15), por lo que llevaremos la cuenta de dichas tres estructuras-f, dibujándolas por separado. Se las muestra en (2.16).

$$(2.16)\quad \begin{array}{l} f_5 \left[PRED \quad \text{`JUGAR} < (f_5\ SUJ)\,(f_5\ OBJ) > \text{'} \right] \\ f_3 \begin{bmatrix} PRED & \text{`DEEPBLUE'} \\ NUM & SING \\ PERS & 3 \end{bmatrix} \quad f_7 \begin{bmatrix} PRED & \text{`AJEDREZ'} \\ NUM & SING \\ PERS & 3 \end{bmatrix} \end{array}$$

Ecuaciones de la forma $f_m = f_n$. Consideremos ahora las ecuaciones de (2.4) que toman la forma de $f_m = f_n$, o sea las ecuaciones listadas en (2.17).

$$(2.17)\quad \begin{array}{ll} \text{(b)} & f_2 = f_3 \\ \text{(f)} & f_1 = f_4 \\ \text{(g)} & f_3 = f_5 \\ \text{(l)} & f_6 = f_7 \end{array}$$

Todas estas sentencias igualan estructuras-f: dicen que las dos variables mencionadas en la ecuación en realidad nombran a la misma estructura-f. Desde otro punto de vista, podemos decir que dos variables igualadas son etiquetas alternativas para una misma estructura-f. En las convenciones gráficas, nombres equivalentes para una estructura-f se escriben uno encima del otro en el lado izquierdo de la estructura-f. Consideremos ahora el ejemplo de descripcion funcional en (2.18)

(2.18)

$$\begin{array}{c} (f_s\ A) = B \\ (f_l\ C) = D \\ f_s = f_l \end{array}$$

Considerando la primera ecuación, podemos fácilmente tomar el primer paso construyendo una estructura-f que satisfaga esta estructura funcional, construyendo la representación en (2.19).

(2.19)

$f_s \, [A \quad B]$

Consideremos ahora la segunda ecuación, la cual dice que hay una estructura-f llamada f_l que contiene una línea donde C es el atributo y D su valor. Pero la última ecuación indica que f_s y f_l nombra la misma estructura-f, un hecho que nos lleva a inferir que la estructura-f en (2.19), conocida alternativamente como f_s ó f_l, tendrá no solamente las características especificadas para f_s, sino también aquellas especificadas para f_l. Por lo tanto, agregamos a (2.19) la línea que la segunda ecuación prescribe para f_l. Podemos registrar el hecho de que (2.19) tiene dos nombres, f_s y f_l escribiendo ambos fuera del corchete izquierdo de la estructura-f. Estas modificaciones se encuentran en (2.20).

(2.20)

$$\begin{array}{l} f_s \\ f_l \end{array} \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Hay otro tema conceptual sobre el cual el lector tendrá que lidiar con respecto a las ecuaciones en (2.17). Recordemos la figura 2.6, la cual era la conceptualización gráfica de la asociación entre nodos de la estructura-c y estructuras-f. Cada nodo se unía con un ícono de estructura-f distinto. Esta representación era una abstracción que dispensaba la pregunta de si cada uno de estos íconos representaba una estructura-f distinta o no. Este podría ser un estado de las cosas completamente posible; sin embargo, no necesita ser éste el caso en que los nodos se ponen en correspondencia uno a uno con estructuras-f. Obviamente en (2.17) estamos diciendo que en nuestro ejemplo particular no es cierto que cada nodo se empareja con una estructura-f distinta. Modificando la figura 2.6 para ser de algún modo menos abstracta con respecto a este tema, podemos dibujar la figura 2.13. Vale la pena estudiar la relación entre los nodos y las estructuras-f en la figura 2.13, de forma de obtener una mejor comprensión de cómo la información funcional desde varios nodos de estructura-c puede ser comprimida en la estructura-f.

Volviendo al tema de la construcción de una estructura-f a partir de la descripción funcional en (2.4), podemos recopilar toda la información acerca de las equivalencias de nombres de estructuras-f tan sólo listando el conjunto completo de nombres de todas las estructuras-f, como en (2.21).

(2.21)

$$\begin{array}{l} f_1 \\ f_2 \\ f_5 \end{array} \begin{bmatrix} PRED & \text{`JUGAR} < (f_5\, SUJ)(f_5\, OBJ) > \text{'} \\ & \\ & \end{bmatrix}$$

$$\begin{array}{l} f_2 \\ f_3 \end{array} \begin{bmatrix} PRED & \text{`DEEPBLUE'} \\ NUM & SING \\ PERS & 3 \end{bmatrix} \quad \begin{array}{l} f_6 \\ f_7 \end{array} \begin{bmatrix} PRED & \text{`AJEDREZ'} \\ NUM & SING \\ PERS & 3 \end{bmatrix}$$

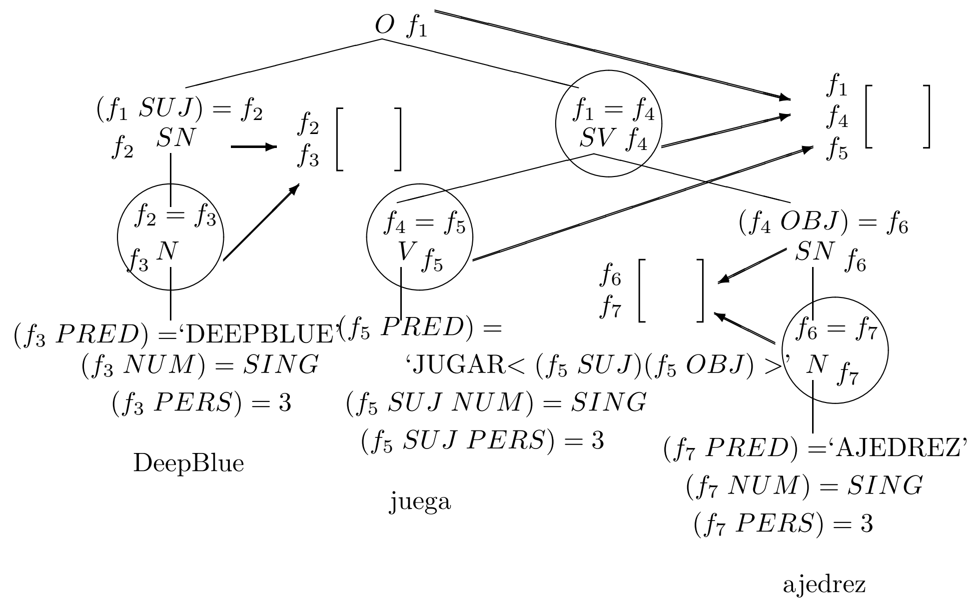


Figura 2.13: Relación revisada entre los nodos en un árbol y las estructuras-f.

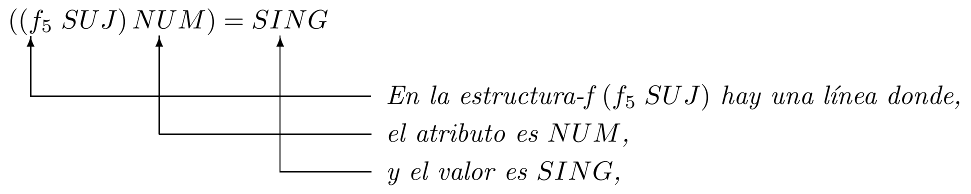


Figura 2.14: Significado de (2.23).

Asociatividad a izquierda. En algunas ecuaciones en (2.4) —específicamente en aquellas en (2.22)— se presentan dos atributos.

(2.22) (i) $(f_5\ SUJ\ NUM) = SING$
(j) $(f_5\ SUJ\ PERS) = 3$

Tomemos (2.22)(i) como objeto de nuestro análisis. Para interpretar esta ecuación uno necesita darse cuenta de que la forma de la ecuación vista arriba emplea una convención para abreviar denominada **asociatividad a izquierda:** la forma no abreviada de (2.22)(i) sería como la dada en (2.23).

(2.23)

$((f_5\ SUJ)\ NUM) = SING$

Podemos ahora simplemente aplicar el esquema de interpretación para ecuaciones funcionales (lo visto más arriba).

Por lo tanto la expresión (f_5 SUJ) designa una estructura-f, tal como se puede inferir a partir del significado de (2.23) del modo que está desglosado en la figura 2.14. En otras palabras, la estructura-f f_5 debe contener en ella una línea donde el atributo sea SUJ. Es más, el valor encontrado en dicha línea debe ser una estructura-f, y dicha estructura-f debe tener las características que se detallan en (2.23). Por lo que las modificaciones que debemos hacer a las estructuras-f que hemos estado haciendo hasta (2.21) se muestran en (2.24).

(2.24)
$$\begin{array}{l} f_1 \\ f_2 \\ f_5 \end{array}\begin{bmatrix} PRED & \text{`JUGAR} < (f_5\ SUJ)(f_5\ OBJ) >\text{'} \\ SUJ & [NUM \quad SING] \\ & \end{bmatrix}$$
$$\begin{array}{l} f_2 \\ f_3 \end{array}\begin{bmatrix} PRED & \text{`DEEPBLUE'} \\ NUM & SING \\ PERS & 3 \end{bmatrix} \quad \begin{array}{l} f_6 \\ f_7 \end{array}\begin{bmatrix} PRED & \text{`AJEDREZ'} \\ NUM & SING \\ PERS & 3 \end{bmatrix}$$

Este es un lugar oportuno para resaltar un detalle respecto de la forma (f_n ATR), tal como en (f_5 SUJ). Recordemos el hecho que las estructuras-f están sujetas a una restricción de unicidad tal que una estructura-f no puede contener dos lineas que tengan el mismo atributo pero distinto valor (Subsección 2.5.1). Como una consecuencia de dicha restricción, un valor dado puede ser unívocamente identificado mencionando la estructura-f en la que se encuentra y el atributo con el cual está puesto en el par. De este modo, una expresión de la forma (f_p ATR) unívocamente nombra algo; i.e., el valor que está en pareja con ATR en la estructura-f f_p. Así, (f_5 SUJ) nombra o designa unívocamente la estructura-f que añadimos a f_5 en (2.24), y cualquier uso subsecuente de la expresión (f_5 SUJ) se referirá a la misma estructura-f.

Ahora podemos pasar a la siguiente ecuación en (2.22), a la cual, aplicando el mismo proceso que utilizamos para tratar con la ecuación anterior escribimos (2.22) sin abreviar en (2.25)

(2.25)

$((f_5\ SUJ)\ PERS) = SING$

Ahora sabemos que (f_5 SUJ) designa a una estructura-f en particular, y simplemente agregamos a dicha estructura-f una línea con $PERS$ como atributo y 3 como valor, como en (2.26).

(2.26)
$$\begin{array}{l} f_1 \\ f_2 \\ f_5 \end{array}\begin{bmatrix} PRED & \text{`JUGAR} < (f_5\ SUJ)(f_5\ OBJ) >\text{'} \\ SUJ & \begin{bmatrix} NUM & SING \\ PERS & 3 \end{bmatrix} \\ & \end{bmatrix}$$
$$\begin{array}{l} f_2 \\ f_3 \end{array}\begin{bmatrix} PRED & \text{`DEEPBLUE'} \\ NUM & SING \\ PERS & 3 \end{bmatrix} \quad \begin{array}{l} f_6 \\ f_7 \end{array}\begin{bmatrix} PRED & \text{`AJEDREZ'} \\ NUM & SING \\ PERS & 3 \end{bmatrix}$$

Ahora nos faltan de examinar tan sólo dos ecuaciones en la descripción funcional para la oracion (2.3). Éstas son listadas a continuación.

(2.27) (a) $(f_1\ SUJ) = f_2$
(b) $(f_4\ OBJ) = f_6$

La segunda de estas ecuaciones puede ser resuelta de una forma rápida y sencilla. Esta ecuación dice que la estructura-f f_4 contiene una línea donde el atributo OBJ está emparejado con la estructura-f f_6. Por lo tanto, simplemente ponemos f_6, que hemos estado construyendo, dentro de f_4 dentro del atributo OBJ en una forma ilustrada como sigue.

(2.28)
$$\begin{array}{l} f_1 \\ f_2 \\ f_5 \end{array} \begin{bmatrix} PRED & \text{`JUGAR} < (f_5\ SUJ)(f_5\ OBJ) > \text{'} \\ SUJ & \begin{bmatrix} NUM & SING \\ PERS & 3 \end{bmatrix} \\ OBJ & \begin{array}{l} f_6 \\ f_7 \end{array} \begin{bmatrix} PRED & \text{`AJEDREZ'} \\ NUM & SING \\ PERS & 3 \end{bmatrix} \end{bmatrix}$$
$$\begin{array}{l} f_2 \\ f_3 \end{array} \begin{bmatrix} PRED & \text{`DEEPBLUE'} \\ NUM & SING \\ PERS & 3 \end{bmatrix}$$

La última ecuación dice que el valor que está emparejado con SUJ en f_1 es f_2. Sin embargo, ahora parece que hay un valor ya existente para SUJ en f_1, y que es la estructura-f mostrada en (2.29).

(2.29)

$$\begin{bmatrix} NUM & SING \\ PERS & 3 \end{bmatrix}$$

Esto significa que la estructura-f en (2.29) y aquella en f_2 son de hecho representaciones (parciales) de la misma cosa. En este caso el valor de SUJ es la **mezcla** de ambas estructuras-f.

Mezcla:

La mezcla de dos instancias de un mismo nombre atómico consiste en dicho nombre atómico. Los nombres atómicos que no son idénticos no se mezclan. Las formas semánticas (encerradas entre por '...') *nunca* se mezclan. A los efectos de mezclar dos estructuras-f —llamémoslas f_m y f_n— elegimos una de éstas, por ejemplo f_m, y para cada atributo a en f_m, tratamos de encontrar una instancia de a en f_n. Llamemos al valor asociado con a en f_m, v. Si a no ocurre en f_n, entonces agregamos el atributo a y el valor v a f_n. Por el contrario, si a ya está presente en f_n, y su valor es v', entonces la mezcla de v y v' será el nuevo valor de a en f_n. Si todas las mezclas subsidiarias son exitosas, entonces la versión modificada de f_n representa la mezcla de f_m y f_n.

Atr. \ Estr.-f	f_2		(2.29)		MEZCLA
$PRED$	'DEEPBLUE'		$\emptyset$	$\longrightarrow$	'DEEPBLUE'
NUM	$SING$	$\longleftrightarrow$	$SING$	$\longrightarrow$	$SING$
$PERS$	3	$\longleftrightarrow$	3	$\longrightarrow$	3

Figura 2.15: Mezcla de f_2 y de (2.29).

Si la mezcla de estas estructuras-f fallara, entonces no habría ninguna estructura-f válida para descripción-f en (2.4), y la gramáctica estaría prediciendo que la oración es agramatical. Sucede en este caso que las estructuras-f se mezclan de manera exitosa: la figura 2.15 esquematiza el proceso de construcción. Finalmente, la estructura-f terminada aparece en (2.30).

$$(2.30) \qquad \begin{array}{c} f_1 \\ f_4 \\ f_5 \end{array} \left[\begin{array}{ll} PRED & \text{'JUGAR} < (f_5\ SUJ)(f_5\ OBJ) > \text{'} \\ SUJ & \begin{array}{c} f_2 \\ f_3 \end{array} \left[\begin{array}{ll} PRED & \text{'DEEPBLUE'} \\ NUM & SING \\ PERS & 3 \end{array} \right] \\ OBJ & \begin{array}{c} f_6 \\ f_7 \end{array} \left[\begin{array}{ll} PRED & \text{'AJEDREZ'} \\ NUM & SING \\ PERS & 3 \end{array} \right] \end{array} \right]$$

2.6 Estructuras-f y la gramaticalidad de las oraciones

Para que una Gramática Léxico Funcional prediga que una oración es gramatical, ésta debe satisfacer dos criterios. En primer lugar, debe ser posible para la gramática asignarle un árbol de estructura de constituyentes, y, en segundo lugar, la gramática debe asignarle una estructura-f bien formada. El primer criterio es familiar a cualquier lingüista que tenga algún entrenamiento en sintaxis. Para el segundo, significa que uno debe ser capaz de construir una estructura-f para la descripción funcional obtenida a partir de la estructura-c totalmente instanciada y también que dicha estructura-f debe estar de acuerdo con ciertos principos de formación de las estructuras-f.

2.6.1 Consistencia.

La consistencia es una propiedad de las descripciones funcionales: una descripción funcional es consistente si existe una estructura-f que pueda hacer todas las ecuaciones en esa descripción funcional verdaderas. Este punto merece elaboración. Recordemos que hemos establecido más arriba que una estructura-f puede asociar a lo más un valor con un atributo dado. Visto este hecho debe ser obvio que no toda colección de ecuaciones funcionales puede determinar una estructura-f que adhiera a la condición de unicidad. Para empezar, las dos ecuaciones listadas aquí son incompatibles.

$$(2.31) \qquad \begin{array}{ll} \text{(a)} & (f_{43}\ PERS) = 2 \\ \text{(b)} & (f_{43}\ PERS) = 3 \end{array}$$

Tomadas juntas, implican que ambos valores 2 y 3 deben ser asignados al atributo $PERS$ en f_{43}, por lo tanto violándose la condición de unicidad. Cuando sea imposible construir una estructura-f a partir de una descripción funcional, decimos que dicha descripción funcional es **inconsistente.**

2.6.2 Completitud y coherencia.

Paso seguido consideramos condiciones referidas a estructuras-f terminadas. Hasta este punto hemos estado pensando acerca estructuras-f en un forma un poco mecánica; ahora, sin embargo, es tiempo de alejarse un poco y reflexionar acerca de qué tipos de conceptos lingüísticos se supone que una estructura-f expresa. Básicamente, las estructuras-f son un repositorio de toda información acerca de funciones gramaticales, tales como *sujeto, objeto,* etcétera. En LFG se piensa a las funciones gramaticales como partes primitivas de la sintaxis natural del lenguaje, esto es, no derivables desde ningún otro aspecto de la gramática. En otras palabras, parece haber generalizaciones acerca de regularidades en sintaxis oracional, morfología, relaciones parafrasales, etcétera las cuales no son completamente expresables cuando nos limitamos al aparato disponible para distinguir entre elementos dentro de nociones clausales tales como la de *caso* (por ejemplo, *P nominativa)*, *estructura de constituyentes* (por ejemplo, *SN dominado inmediatamente por S*), o de *argumentación semántica* (por ejemplo, *agente* en un sistema de semántica que distingue entre tipos de argumentos). Nociones tradicionales de funciones gramaticales (por ejemplo *sujeto*) aparecen, sin embargo, para proveer una clasificación útil de los componentes clausales, lo cual facilita la formulación de ciertas generalizaciones interesantes para la lingüística, sobre las cuales las nociones mencionadas anteriormente no proveen muchas pistas. Argumentos en favor de esta postura abundan en la literatura sobre LFG. Consecuentemente uno de las principales motivaciones detrás de la LFG ha sido el deseo de construir una teoría de las funciones gramaticales, y las estructuras-f se han empleado como un medio de mostrar relaciones entre estas funciones.

Considerese la estructura-f terminada para el ejemplo (2.3) delineada en (2.30). La estructura-f externa es la que corresponde a toda la oración. (Puede ser útil referirse también a la figura 2.13). Dentro de esta estructura-f se encuentran líneas donde los atributos son SUJ, OBJ, y $PRED$. Podría haber, por supuesto, un conjunto totalmente diferente de atributos, si estuviemos tratando con otra estructura. Estos nombre son, obviamente, abreviaciones transparentes para *sujeto, objeto* y *predicado*, todos conceptos familiares usados en el trabajo con gramáticas de lenguajes naturales. Así no hay ninguna sorpresa en que los valores de estos atributos sean interpretados como las representaciones de estas características lingüísticamente relevantes.

En esta sección vamos a discutir restricciones en las estructuras-f que son esencialmente clausulas acerca de las relaciones entre las funciones gramaticales manifestadas en la oración y en el predicado principal de la oración. Una de las suposiciones principales en LFG es que la acción de las funciones gramaticales está regulada a través de lo que se denomina argumento de predicado, el cual se encuentra en la forma semántica emparejada con $PRED$. Las formas semánticas aparecen gráficamente como material encerrado entre comillas sencillas.

(2.32)
(a) 'DEEPBLUE'
(b) 'JUGAR$< (\uparrow\ SUJ)(\uparrow\ OBJ) >$'

Todas las formas semánticas contienen una expresión, por ejemplo, DEEPBLUE o JUGAR, la cual se interpreta en la semántica, y algunas incorporan **estructuras de argumentos de predicado,** i.e., listas de argumentos los cuales expresan relaciones sobre las entidades semánticas. Estas listas de argumentos están encerradas entre paréntesis en ángulo, y cada posición entre estos paréntesis significa un argumento semántico.

Ahora, vamos a resaltar un tema obvio, los argumentos en la estructura de argumentos de predicado son elementos empleados en el nivel de representación semántica. La gramática debe expresar cómo se relaciona el nivel de representación sintáctica con el de la semántica, y las entidades sintácticas para las cuales LFG asocia argumentos semánticos son funciones gramaticales. La asociación de funciones gramaticales con argumentos semánticos se establece almacenando los nombres de las funciones gramaticales en posiciones de argumentos dados (ver (2.32)(b)). Cuando una función gramatical está asociada con un argumento semántico, se dice que está **gobernada** por este argumento. Es a través de este ligamiento de argumentos y funciones gramaticales que LFG soporta muchas relaciones parafrasales; los principios que gobiernan variaciones en estas asociaciones y sus diversas consecuencias morfológicas y semánticas son el centro de la mayor parte de la investigación en LFG. La unión de funciones gramaticales con los argumentos determina también en un sentido más amplio qué estructuras pueden o no ocurrir como elementos de una oración. Es lo mismo que decir que estas estructuras de argumentos de predicado refuerzan restricciones de subcategorización. Esto es posible por la existencia de dos principios conocidos como **completitud** y **coherencia.**

Completitud significa que si una función gramatical es mencionada en la estructura de argumentos de predicado, entonces debe estar presente en la estructura-f. El principio de completitud tiene en cuenta la desviación de ejemplo como los vistos en (2.33).

(2.33)

Juan gusta.

Asumiendo una entrada léxica para *gusta* que contenga las siguientes oraciones.

$$(2.34)\qquad \begin{array}{lll} \text{gusta} & V & (\uparrow PRED) = \text{`GUSTA} < (\uparrow SUJ)(\uparrow OBJ) >\text{'} \\ & & (\uparrow SUJ\ NUM) = SING \\ & & (\uparrow SUJ\ PERS) = 3 \end{array}$$

Es posible verificar que (2.35) es la estructura-f que nuestra gramática de ejemplo aumentada con el ítem léxico de (2.34), nos daría para (2.33).

$$(2.35)\qquad \begin{bmatrix} PRED & \text{`GUSTA} < (\uparrow SUJ)(\uparrow OBJ) >\text{'} \\ SUJ & \begin{bmatrix} PRED & \text{`JUAN'} \\ NUM & SING \\ PERS & 3 \end{bmatrix} \end{bmatrix}$$

Esta estructura-f no muestra ninguna variable nombrando sus componentes, por lo que hemos adoptado la fácilmente interpretable convención de retener las metavariables madre dentro de las formas semánticas tales como *gusta.* Los referentes de estas metavariables serán la estructura-f dominante inmediatamente. Como (2.35) no provee un valor para OBJ, se viola completitud. Nótese que para que haya un valor para OBJ en (2.35), la oración correspondiente debería haber tenido un SN después del verbo: una mirada a nuestra gramática de ejemplo confirmará esto. Por lo tanto, uno puede fácilmente ver como la completitud asume el rol positivo de la subcategorización, i.e. el de asegurar la presencia de ciertos elementos.

La coherencia puede ser vista como la inversa de la completitud. Este principio limita la ocurrencia de funciones gramaticales gobernables dentro de estructuras-f. Si alguna entrada léxica en la gramática menciona a una función gramatical dada en la estructura de argumentos de predicado, entonces esta función gramatical es gobernable. La coherencia establece que si una función gramatical gobernable ocurre en una estructura-f, entonces debe ser ciertamente gobernada por algún argumento en los argumentos de predicado del predicado de esa estructura-f. Esto obviamente completa el rol negativo de la subcategorización, la exclusión de elementos no específicamente habilitados por el predicado. Así, dada una entrada léxica tal como en (2.36), la oración de (2.37) dará una estructura-f como en (2.38).

(2.36)
$$\begin{array}{lll} \text{cae} & V & (\uparrow PRED) = \text{`CAE} < (\uparrow SUJ) >\text{'} \\ & & (\uparrow SUJ\ NUM) = SING \\ & & (\uparrow SUJ\ PERS) = 3 \end{array}$$

(2.37)

Juan cae María.

$$(2.38)\quad \begin{bmatrix} PRED & \text{`CAE} < (\uparrow SUJ) >\text{'} \\ & \begin{bmatrix} PRED & \text{`JUAN'} \\ & SING \\ PERS & 3 \end{bmatrix} \\ & \begin{bmatrix} PRED & \text{`MARÍA'} \\ & SING \\ PERS & 3 \end{bmatrix} \end{bmatrix}$$

La estructura-f en (2.38) muestra una estructura de argumentos de predicado donde no hay ninguna mención al objeto. Esto expresa de manera apropiada el problema obvio con (2.37): ningún objeto, *María,* puede ser gobernado por un argumento semántico y es, por tanto, superfluo.

2.7 Ecuaciones restrictivas y valores booleanos

En esta sección final vamos a señalar la existencia de algunas características adicionales de las ecuaciones funcionales.

2.7.1 Ecuaciones restrictivas.

Las ecuaciones restrictivas se identifican por la letra c en el símbolo $=_c$. Respecto de los aspectos mecánicos involucrados en la utilización de este dispositivo, uno comienza particionando la descripción funcional en dos conjuntos, uno conteniendo sólo ecuaciones restrictivas, y otro con las ecuaciones no-restrictivas. Una vez hecho esto, uno considera el conjunto de ecuaciones no-restrictivas y construye la mínima estructura-f que éste especifica. Solamente después que la estructura-f ha sido construida se ponen en uso las ecuaciones restrictivas. Si todas las ecuaciones restrictivas son hechas ciertas por la estructura-f construida a partir de las ecuaciones no-restrictivas, entonces la estructura-f estará bien formada. De otro modo es inconsistente.

Los siguientes ejemplos ilustran el uso de las ecuaciones restrictivas. Considérense las siguiente descripción funcional.

$$(2.39)\quad \begin{array}{ll} \text{(a)} & (f_n\ A) = B \\ \text{(b)} & (f_n\ C) = D \\ \text{(c)} & (f_n\ C) =_c D \end{array}$$

Primero examinamos las ecuaciones no restrictivas en (2.39), esto es (2.39)(a) y (2.39)(b). La estructura funcional mínima que estas ecuaciones describen está en (2.40).

(2.40)

$$f_n \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Nótese que (2.40) es la estructura-f completa para la descripción funcional en (2.39): la ecuación (2.39)(c) es cierta en (2.40), esto es, f_n contiene una línea que empareja al atributo C con el valor D. Como f_n tiene la característica deseada, la ecuación restrictiva es satisfecha y la estructura-f es coherente.

2.7.2 Negativos.

Las ecuaciones funcionales pueden tener un operador negativo. Éste puede ser escrito como sigue.

(2.41)

$\neg(f_n\ E) = F$

La ecuación resultante es una variante de la ecuación restrictiva. Esto es, también se aplica luego de que se hayan considerado todas las ecuaciones normales y la estructura-f mínima esté completa. Sin embargo, una ecuación negativa, digamos $\neg eq$ es satisfecha por una estructura-f dada, sólo en el caso de que eq sea falso respecto de dicha estructura-f. Esta característica se ilustra más abajo.

Asumamos la siguiente descripción funcional.

	(a)	$(f_n\ A) = B$
(2.42)	(b)	$(f_n\ C) = D$
	(c)	$\neg(f_n\ E) = F$

Usando (2.42)(a) y (2.42)(b) construimos (2.43).

(2.43)

$$f_n \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Fijándonos ahora en (2.42)(c) consideramos ahora su contrario, (2.44)

(2.44)

$(f_n\ E) = F$

Sucede que (2.44) es falso con respecto a (2.43) y así haciendo (2.42)(c) verdadera. Por lo tanto (2.42) es consistente.

2.7.3 Conjunción.

Hemos estado empleando tácitamente conjunción a través de toda la mitad anterior del capítulo. En las descripciones-f, como en (2.4), hay típicamente muchas ecuaciones, y todas ellas deben ser satisfechas. Esta es la esencia de la conjunción: la conjunción de las ecuaciones $eq_1, \ldots, eq_n$ es cierta, precisamente cuando cada una de las ecuaciones que componen $eq_1, \ldots, eq_n$ es cierta por separado. Por lo tanto, ahora ponemos de forma explícita el hecho de que asumimos una conjunción siempre que se escribe una secuencia de dos o más ecuaciones. A veces se hace necesario sin embargo quitar la ambigüedad en expresiones involucrando a más de un valor booleano. Por ejemplo, supongamos que uno ve (2.45): ¿la negación se aplica tan sólo a la primera de las ecuaciones o la conjunción de ambas?

(2.45)
$$\begin{array}{l} \neg(f_n\ A) = B \\ \ (f_n\ C) = D \end{array}$$

Debemos establecer una convención para decidir sobre tales cuestiones. Sigamos la práctica tradicional de asumir que la negación se aplica a la entidad más pequeña inmediatiamente a la derecha. Esto significa en el caso de (2.45) que sólo la primera ecuación está negada. De modo de indicar que la negación de la conjunción es lo que se quiere decir, podemos emplear alguna de las convenciones de paréntesis de (2.46)(a) y (b).

(2.46)
(a) $$\neg \begin{bmatrix} (f_n\ A) = B \\ (f_n\ C) = D \end{bmatrix}$$
(b) $$\neg\,[\,(f_n\ A) = B \quad (f_n\ C) = D\,]$$

2.7.4 Disyunción.

La última construcción booleana utilizada en LFG es la disyunción. Ésta une dos o más ecuaciones, como en caso del operador conjunción. Una disyunción como en (2.47) será verdadera si alguna o varias de las ecuaciones que la componen sea verdadera.

(2.47)

$$\left\{ \begin{array}{l} (f_n\ A) = B \\ (f_n\ C) = D \end{array} \right\}$$

Sin embargo, la naturaleza de las estructuras-f determina ciertas consecuencias inesperadas.

La pregunta que debemos formularnos es qué significa decir que una o ambas de las ecuaciones que componen la disyunción debe ser cierta. De hecho, esto significa que las descripciones funcionales que contienen disyunciones, pueden corresponder a más de una estructura-f. Para ver este punto tomemos la siguiente descripción funcional como ejemplo.

$$(2.48)\qquad \begin{array}{ll} \text{(a)} & \left\{ \begin{array}{l} (f_n\ A) = B \\ (f_n\ C) = D \end{array} \right\} \\ \text{(b)} & (f_n\ E) = F \end{array}$$

Nótese que hay tres candidatas obvias para que ser la estructura-f que haga las dos ecuaciones (2.48) ciertas.

$$(2.49)\qquad \text{(a)}\quad f_n \begin{bmatrix} A & B \\ E & F \end{bmatrix} \qquad \text{(b)}\quad f_n \begin{bmatrix} C & D \\ E & F \end{bmatrix} \qquad \text{(c)}\quad f_n \begin{bmatrix} A & B \\ C & D \\ E & F \end{bmatrix}$$

Todas estas estructuras-f hacen (2.48)(b) verdadera, al contener una línea en la cual el atributo es E y el valor F. La estructura-f en (2.49)(a) hace (2.48)(a) cierta, pues la primera línea hace válida la primera disyuntiva (ecuación componente de una disyunción) de (2.48)(a), haciendo verdadera toda la disyunción. De forma similar, la primera línea de (2.49)(b) hace la seguna disyuntiva de (2.48)(a) cierta. Finalmente, notemos que las primeras dos líneas de (2.49)(c) hacen válidas ambas disyuntivas en (2.48)(a). En resumen todas estas estructuras-f satisfacen la descripción funcional en (2.48).

Consideremos ahora las tres estructuras-f en (2.49) en términos de minimalidad. Comenzando con (2.49)(a), notamos que sacar la primera línea invalida (2.48)(a), y que sacar la segunda invalida (2.48)(b). Como ninguna línea puede ser extraída, (2.49)(a) es minimal. Pasamos a la segunda estructura-f, también podemos verificar que ninguna de sus dos líneas puede ser removida sin invalidar uno u otra de las ecuaciones en (2.48). Por lo tanto (2.49)(b) es también minimal. Finalmente, consideremos (2.49)(c). Fijémonos que aquí si sustraemos cualquiera (no ambas) de las dos primeras líneas de la estructura-f ambas ecuaciones en (2.48) seguirán siendo ciertas. Diremos entonces que dicha estructura-f no es minimal y por lo tanto no estará licenciada por (2.48).

Finalmente hagamos explícito que las ecuaciones dentro de una disyunción serán interpretadas siempre como disyunciones separadas. Nunca se las trata como una conjunción dentro de una disyunción, a menos que haya una parentización explícita que indique esta interpretación. Por lo tanto, (2.50)(a) contiene tres disyunciones, y (2.50)(b), dos.

$$(2.50)\qquad \begin{array}{ll} \text{(a)} & \left\{ \begin{array}{l} (f_n\ A) = B \\ (f_n\ C) = D \\ (f_n\ E) = F) \end{array} \right\} \\ \text{(b)} & \left\{ \begin{array}{c} (f_n\ A) = B \\ \begin{bmatrix} (f_n\ C) = D \\ (f_n\ E) = F) \end{bmatrix} \end{array} \right\} \end{array}$$

Capítulo 3

Análisis Sintáctico

En este capítulo vamos a centrarnos en la generación del árbol de constituyentes (estructura-c) del proceso de creación de las estructura-f. Para ello comenzaremos estudiando técnicas de análisis sintáctico para lenguajes libres de contexto tales como los definidos mediante las reglas de LFG. De manera preliminar, tendremos algunos comentarios respecto de la utilidad de los lenguajes libres de contexto para la lingüística computacional, pues siguiendo [12] y [24] la literatura de NLP no le ha prestado por mucho tiempo la importancia que merecen este tipo de lenguajes. Luego se estudiarán muy brevemente las técnicas clásicas de análisis sintáctico para CFGs. El núcleo de este capítulo y, aun más, de este trabajo, lo constituyen los combinadores monádicos de *parsing*. Éste será el objeto de la última sección.

3.1 Lenguajes libres de contexto

Siguiendo a [12] en la siguiente sección vamos a abordar el siguiente interrogante: ¿Qué tan grande debe ser el poder de expresión —a nivel formal— de una gramática que analice lenguaje natural? Estamos hablando de *parsing* por lo que de uno u otro modo estamos involucrando una gramática. Esta pregunta puede ser reformulada en estos términos: ¿Cuál es la más pequeña clase conocida de lenguajes formales que puede razonablemente considerarse como conteniendo a todos los lenguajes naturales?

3.1.1 Lenguajes finitos

Obviamente hay una vasta cantidad de construcciones que prueban que la cantidad de oraciones en un lenguaje natural no puede ser finita. De hecho no se conoce ningúna lengua humana que sea finita. Para citar alguna de dichas construcciones tomemos por ejemplo el caso de la coordinación, que permite una cantidad no acotada de conjunciones. Y el español nos permite iterar los adjetivos indefinidamente (una casa linda, ilumindada, fresca, ...) así también como las oraciones relativas que a su vez contienen sintagmas verbales que pueden contener sintagnas nominales que pueden contener oraciones relativas las cuales...

3.1.2 Lenguajes de estado finito

La aseveración de Chomsky en [9] de que los lenguajes naturales no son en general de estado finito es correcta. El siguiente argumento es una demostración válida para el inglés, el conjunto (3.1):

(3.1)

{A white male (whom a white male)^n (hired)^n hired another white male. $| \; n > 0$}

es la intersección del inglés con el conjunto regular (3.2):

(3.2)

A white male (whom a white male)* hired* another white male.

(En términos gramaticales comunes, esto sucede pues cada ocurrencia de la frase *white male* es un sintagma nominal el cual necesita una verbo tal como *hired* para completar una oración de la cual ser sujeto.) Pero (3.1) no es regular; y los conjuntos regulares son cerrados bajo intersección; de este modo el inglés no es regular. *Q.E.D.*

Es perfectamente posible que algunos NLs resulten no tener configuraciones inherentemente auto-insertas que son las que suelen determinar que un lenguaje no sea regular. Los leguajes en los cuales la parataxis es usada mucho más que la hipotaxis (i.e., lenguajes en los cuales las oraciones separadas se presentan linealmente en vez de insertas) son comunes. Sin embargo, podemos esperar configuraciones no-regulares en casi todos los lenguajes del mundo. Es más, hay lenguajes que presentan mejores argumentos de caracter no-regular que el inglés; por ejemplo, hay ciertos fenómenos de inserción central en algunos lenguajes del centro de Sudán que parecen ser más frecuentes que en inglés.

El hecho de que los NLs no son conjuntos regulares es por un lado sorprendente y desilucionante desde el punto de vista del análisis sintáctico. Es sorprendente porque no hay forma más sencilla de obtener lenguajes infinitos que la de admitir las operaciones de concatenación, unión y clausura de Kleene sobre un vocabulario finito, y no hay ninguna razón obvia a priori respecto de porque los humanos no pudieren ser bien servidos por los lenguajes regulares. Y es desilucionante porque si los NLs fueran conjuntos reguales, sabemos que podriamos reconocerlos en tiempo lineal determinista usando el más rápido y más sencillo dispositivo abstracto de cómputo, la máquina de estado finito. Por supuesto, dada cualquier limitación a la memoria finita de una computadora dada, estamos de hecho haciendo tan sólo ésto, pero no es muy revelador desde le punto de vista teóricousar esto como la base de nuestra compresión de la tarea.

3.1.3 Lenguajes libres de contexto deterministas

Los lenguajes de estado finito, por suerte, no son los únicos lenguajes que pueden ser eficientemente reconocidos: hay clases mucho más grandes de lenguajes que tienen reconocimiento en tiempo lineal. Una de tales clases es la de los CFLs deterministas (DCFLs), i.e., aquellos CFLs que son aceptados por algún autómata a pila determinista. Sería razonable, por lo tanto, preguntarse si hay al menos algunos NLs serán DCFLs. Según parece esta pregunta nunca ha sido considerada, menos aun respondida, en la literatura lingüística ni en la de la ciencia de la computación. Abundan los ejemplos en los cuales se hecha por tierra toda la literatura sobre DCFLs, LR *parsing*, y temas relacionados sin siquiera mirarlos sobre la base de argumentos errados (por ejemplo, concordancia sujeto-verbo) que supuestamente demuestran que el inglés no es ni siquiera CFL, por lo tanto, a fortiori tampoco DCFL.

No puede demostrarse que el inglés es ambiguo por el mero hecho de ser ambiguo. La ambigüedad debe ser siempre claramente distinguida de la ambigüedad inherente. Un lenguaje inherentemente ambiguo es aquel en el cual todas las gramáticas que lo generan de forma débil son ambiguas. Las gramáticas LR nunca son ambiguas; pero las gramáticas LR caracterizan exactamente el conjunto de los DCFLs, por lo tanto ningún lenguaje inherentemente ambiguo es un DCFL. Pero parece ser que nunca se ha discutido que el inglés o cualquier otro NL sea inherentemente ambiguo. Pero, obviamente, un DCFL puede tener una gramática ambigua; todos los lenguajes tienen gramáticas ambiguas.

La importancia de esto se pone en evidencia cuando observamos que en las aplicaciones de NLP es frecuente desear que un analizador o un traductor entregue un análisis sencillo de una oración de entrada. Uno puede imaginar un sistema de NL implementado en el cual el lenguaje aceptado está descripto por una CF-PSG ambigua pero que es sin embargo (débilmente) una DCFL.

La idea de un parser determinista con una gramática ambigua, la cual sale directamente de lo que se ha estado haciendo con los lenguajes de programación (por ejemplo el sistema `yacc` [18]) ha sido utilizado para lenguajes naturales en trabajos realizados por Fernando Pereira y Stuart Shieber. Shieber [35] describe una implementación de un analizador sintáctico que utiliza una gramática ambigua pero analiza determinísticamente. El analizador usa shift-reduce scheduling de una forma propuesta por Pereira [31], y utiliza reglas para resolver los conflictos entre las acciones del analizador que son virtualmente las mismas que las dadas para `yacc` por Johnson [18].

Según parece, técnicas tales como parsing LR que vienen directamente de lenguajes de programación y diseño de compiladores (y las cuales tienen mucho mayor variedad e interés formal de lo que se suele reconocer) pueden ser de considerable interés en el contexto de aplicaciones de NLP. Por ejemplo, Tomita [38] utiliza pseudo-paralelismo para extender la técnica LR y obtener mútiples análisis en NLP, y Shieber va inclusive más allá al sugerir implicaciones psicolingüísticas. Es interesante notar que los seres humanos solemos fallar tan mal como el analizador sintáctico de Shieber en ciertos tipos de oraciones que los lingüístas podrín considerar como gramaticales (básicamente, oraciones que fallan de la propiedad de prefijo —esto es, tienen una subcadena propia que es a su vez una oración).

3.1.4 Lenguajes libres de contexto

La creencia de que las CF-PSGs no pueden con la estructura de los NLs, y que por tanto los NLs no son CFLs, está bien difundida. Los libros de texto introductorios de lingüística y otros trabajos con orientación pedagógica han falsamente establecido que un fenómeno tal como es concordancia de sujeto-verbo demuestra que el inglés no es CF. Esto no es así, aun lenguajes de estado finito pueden exhibir dependencias entre símbolos arbitrariamente alejados. Para tomar un ejemplo artificial, supongamos que la última palabra en cada oración tuviese que tener alguna marca especial que fuese determinada por el primer morfema que apareciese en la oración; un autómata finito que aceptara el lenguaje podría simplemente codificar en su información de estado cual fue el morfema de comienzo de la oración y chequear la marca de la última palabra respecto del estado antes de aceptar.

Los trabajos en el campo de la gramática generativa nunca han ofrecido nada que pueda ser tomado seriamente como un argumento de que los NLs no son CFLs. Aun peor, la literatura técnica ofrece un cuarto de siglo de esfuerzos errados al tratar de demostrar que los NLs no son CFLs. Aparte de las falacias concernientes a la concordancia antes mencionadas, se han esgrimido argumentos basados en

1. Construcciones con *respectively* (Bar-Hille y Shamir en 1960; Langendoen en 1977)
2. Oraciones comparativas en inglés (Chomsky en 1963)
3. Incorporación de raices nominales en Mohawk (Postal en 1964)
4. Frases verbales de infinitivo en el holandés (Huybregts 1976)
5. Aseveraciones involucrando expresiones numéricas (Elster en 1978)

Tales esfuerzos errados han continuado. A partir de 1982 se vieron nuevos argumentos basados en

6. Cláusulas *such that* en inglés (Higginbotham en 1984)

7. Cláusulas *sluicing* en inglés (Langendoen y Postal en 1985)

Pero ha sido demostrado que ambos argumentos se basan en falsos supuestos respectos de lo que es gramatical en inglés.

Sin embargo, recientemente se ha encontrado al menos una instancia aparentemente válida de un lenguaje natural con una sintaxis débilmente no-CF. Shieber [36] plantea que los dialectos del alemnán hablado cerca de Zurich, Suiza, muestran evidencias de un patrón en el orden de las palabras en ciertas oraciones subordinadas de infitivo muy similar al observado en el holandés: un número arbitrario de sintagmas nominales (SNs) puede ser seguido por un verbo conjugado y un número específico de verbos en infinitivo, y con relaciones semánticas entre los verbos y los SNs exhibiendo un comportamiento serial cruzado: los verbos más a la derecha en la cadena de verbos toman como su objeto los SNs más a la derecha en la lista de SNs. La subcadena crucial tiene la forma de SN^mV^n. En un caso simple, donde $m = n = 5$, una tal subcadena puede tener un significado como el siguiente

(3.3)

Juan miró a-Pedro dejar a-Pablo ayudar a-José hacer a-Jorge leer.
pero con un orden de las palabras correspondiente a

(3.4)
Juan a-Pedro a-Pablo a-José a-Jorge miró dejar ayudar hacer leer.
SN_1 SN_2 SN_3 SN_4 SN_5 V_1 V_2 V_3 V_4 V_5

Se ha determinado que esta construcción no determina que el holandés sea no-CF. Pero en alemán suizo, a diferencia del holandés, hay una propiedad adicional que hace que este fenómeno sea relevante para una argumentación de conjuntos de cadenas: ciertos verbos demandan caso dativo en vez de acusativo en sus objetos, como una cuestión de pura sintaxis. Este patrón no es uno que en general pueda ser descripto por una CF-PSG. Por ejemplo, si restrinjimos la situación (intersectando con un conjunto regular apropiado) a las oraciones en las cuales todos los SNs acusativos (SN_a) precedan a todos los SNs dativos (SN_d), entonces las oraciones gramaticales así obtenidas serán aquellas donde los verbos que piden acusativo (V_a) precedan a los que requieren dativo (V_d) y con los números que concuerden, esquemáticamente:

(3.5)

$SN_a^m\ SN_d^n\ V_a^m\ V_d^a$
Pero este esquema tiene la forma de un lenguaje como $\{a^mb^nc^md^n|n > 0\}$, el cual no es CF. Shieber presenta argumentos rigurosamente formulados siguiente líneas similares para demostrar que el lenguaje de hecho falla en ser CFL a causa de esta construcción.

Es posible que otros lenguajes también terminen no siendo CF, aunque al presente la configuración de propiedades necesarias parece muy rara. Ciertas propiedades del sueco han dado lugar a dudas en esta dirección, pero ninguna argumentación se ha publicado. También se han mencionado construcciones de reduplicación posiblemente no-CF en la sintaxis del engenni, un lenguaje africano. Alexis Manaster-Ramer ha sugerido que la expresión idiomática en inglés ejemplificada por *RS-232 or not RS-232, this terminal isn't worknig* (donde el patrón *X or no X* es esencial en la aceptación, y *X* puede tomar una cantidad infinita de valores) también ilustra una posibilidad; y deben de haber a su vez muchas otras propiedades en otros lenguajes que son interesantes de ser investigadas en mayor profundidad.

3.1.5 Lenguajes indizados

Los lenguajes indizados (ILs, Aho [1]) son una clase natural de lenguajes formales que forman un superconjunto propio de los CFLs y un subconjunto propio de los lenguajes sensibles al contexto. Se ha demostrado que esta clase comprende algunos lenguajes NP-completos. Son de interes en el contexto actual porque no hay fenómeno conocido que pueda llevar a uno a creer que los NLs puedan caer fuera de su dominio. En particular, está claro que hay gramáticas indizadas para los hechos del alemán suizo y para casi todos los otros conjuntos de hechos que han sido alguna vez conjeturados como teniendo problemas para una descripción en CF (pero puede llegar a haber problemas aun más difíciles, relacionados más con la semántica que con la sintaxis).

Los lenguajes indizados nos proveen, entonces, al menos por el momento de algún tipo de cota superior para fenómeno sintáctico. Ya no podemos mostrarnos sorprendidos por patrones no-CFL (aunque su rareza es una cuestión de cierto interés) pero deberíamos sorprendernos mucho, e inclusive ser muy suspicaces respecto de supuestos fenómenos no-IL.

3.1.6 Más allá de los lenguajes indizados

Como acabamos de indicar, no parece haber actualmente hechos conocidos que determinen que uno pueda creer que los NLs caen fuera de los ILs, y ante la ausencia de tales hechos, la conclusión conservadora es que los NLs caen dentro de los ILs. Sin embargo algunos autores postulan que los lenguajes naturales no son conjuntos recursivamente enumerables. Pero los argumentos esgrimidos normalmente no son formales o son muy dependientes de alguna determinada teoría.

3.2 Combinadores monádicos para análisis sintáctico

En esta sección vamos a analizar una poderosa herramienta en la programación en lenguajes funcionales puros de evaluación perezosa: Las mónadas y su aplicación en la creación de analizadores sintácticos por el método del descenso recursivo. Este método, si bien no es para nada el más eficiente (véase [24]) presenta muchas ventajas referidas a sus sencillez, facilidad de implementación y comprensión. Comenzaremos, entonces, estudiando las mónadas (en general, siguiendo [39, 40, 41]), para concluir con combinadores de análisis sintáctico monádicos (siguiendo a [16, 17]).

3.2.1 Mónadas

Historia

De la rama de la teoría de categorías surgen las *mónadas* en los años 60 [25, 23] para expresar de manera concisa ciertos aspectos del álgebra universal. Aparece primero en el área del álgebra homológica pero más tarde se le reconocen aplicaciones mucho más amplias (gracias al trabajo de Kleisli y de Eilenberg y Moore). Su importancia emerge lentamente: en los primeros tiempos, ni siquiera tenían un nombre apropiado, sino que se las denominaba simplemente una "construcción estandar" o un "triple". Las formas utilizadas en programación funcional se deben a Kleisli.

Eugenio Moggi propuso a las mónadas como una herramienta estructural útil para la semántica denotacional [28, 29]. Él mostró como al lambda cálculo puede dársele una semántica de *call-by-value* y *call-by-name* en una mónada arbitraria, y como las mónadas podían encapsular una gran variedad de características de los lenguajes de programación, tales como estado, manejo de excepciones y continuaciones.

Independientemente de Moggi, y más o menos al mismo tiempo, Michael Spivey propuso que las mónadas proveen una herramienta estructural útil para el manejo de excepciones en un lenguaje funcional puro y demostró su tesis con un elegante programa de reescritura de términos [37]. Él mostró como las mónadas podían, entonces, tratar excepciones y elección nodeterminista en un marco de trabajo común.

Wadler, inspirado en Moggi y Spivey, propuso las mónadas como una técnica general para estructurar programas funcionales. Sus primeras propuestas tomaban un enfoque especial para la sintaxis de la mónadas (*mónadas por comprensión*, por similitud con definición de listas por comprensión [39]). Esta idea fue poco afortunada pues llevó a muchos a pensar que era necesaria una sintaxis especial. Sin embargo esto no es así, como queda de manifiesto en sus posteriores trabajos ([40, 41]).

Definición más categórica

Veamos primero una definición de mónada más cercana a la teoría de categorías; una mónada, entonces, consiste en considerar un operador M sobre tipos junto con las siguientes tres funciones:

(3.6)

$$\begin{array}{lcl} map & :: & (x \to y) \to (Mx \to My), \\ unit & :: & x \to Mx, \\ join & :: & M(Mx) \to Mx, \end{array}$$

Tales que se verifican las siguientes propiedades:

$$\begin{array}{llrcl} & \text{(i)} & map\, id & = & id, \\ & \text{(ii)} & map\,(g \cdot f) & = & map\, g \cdot map\, f, \\ & \text{(iii)} & map\, f \cdot unit & = & unit \cdot f, \\ (3.7) & \text{(iv)} & map\, f \cdot join & = & join \cdot map\,(map\, f), \\ & \text{(I)} & join \cdot unit & = & id, \\ & \text{(II)} & join \cdot map\, unit & = & id, \\ & \text{(III)} & join \cdot join & = & join \cdot map\, joinf \end{array}$$

En términos categóricos, *unit* y *join* son *transformaciones naturales*. En vez de tratar a *unit* como una sola función de tipo polimórfico, los categóricos la tratan como una familia de flechas, $unit_x :: x \to M\ x$, una para cada objeto x, satisfaciendo $map\, f \cdot unit_x = unit_y \cdot f$ para cualesquiera objetos x y y y cualquier flecha $f :: x \to y$ entre ellos. De igual modo tratan a *join*. Las transformaciones naturales son un concepto más sencillo que el de función polimórfica, pero utilizaremos polimorfismo pues es más familiar a los programadores funcionales.

Este concepto de mónada es un poco más fuerte que áquel que un categórico denotaría con tal nombre: estamos en realidad usando lo que un categórico llamaría una *mónada fuerte* en una *categoría cartesiana cerrada*. A grandes razgos, una categoría es cartesiana cerrada si tiene suficiente estructura para interpretar λ-cálculo. En particular, asociado a cualquier par de objetos (tipos) x y y hay un objeto $[x \rightarrow y]$ representando el espacio de todas las flechas (funciones) desde x a y. Entonces M es un functor si para cualquier flecha $F :: x \rightarrow y$ existe una flecha $map\ f :: M\ x \rightarrow M\ y$ tal que satisfaga (3.7)(i) y (ii). Dicho functor es fuerte si está él mismo representado por una sola flecha $map :: [x \rightarrow y] \rightarrow [M\ x \rightarrow M\ y]$. Esto es evidente para un programador de lenguajes funcionales, pero un categórico sólo provee tanta estructura cuando es necesaria.

Definición

Sin embargo es más útil para nuestros propósitos, considerar que una *mónada* será una tres-upla (`M`, `unitM`, `bindM`) consistente en un constructor de tipos `M` y dos funciones polimórficas.

(3.8)

```
unitM  ::  a -> M a,
bindM  ::  M a -> (a -> M b) -> M b,
```

Estas funciones deben satisfacer tres leyes:

Identidad a izquierda:	`(unitM a) 'bindM' k = k a`
Identidad a derecha:	`m 'bindM' unitM = m`
Asociatividad:	`m'bindM'(\a->(k a)'bindM'(\b->h b))` `= (m'bindM'(\a->k a)'bindM'(\b->h b)`

Estas leyes garantizan que la composición monádica sea asociativa y tenga una identidad a derecha y a izquierda.

Ambas definiciones son equivalentes, pues puedo definir una en función de la otra, de forma tal que las propiedades (3.7) y 3.2.1 se implican unas a otras y viceversa:

(3.9)

```
map f m      = m 'bindM' (\a -> unitM(f a))
join z       = z 'bindM' (\m -> m)

m 'bindM' k  = join (map k m)
```

3.2.2 Mónadas con un *cero* y un *más*

Una extensión que nos es interesante (pues se utiliza para la realización de *parsers*) es cuando podemos definir sobre la mónada las siguientes dos operaciones:

(3.10)

```
zeroM  ::  M a,
plusM  ::  M a -> M a -> M a,
```

de forma tal que se satisfagan las siguientes propiedades:

(3.11)

```
        zeroM 'plusM' p  = p
        p 'plusM' zeroM  = p
p 'plusM' (q 'plusM' q)  = (p 'plusM' q) 'plusM' r
```

Es decir, `zeroM` es una identidad a izquierda y derecha de `plusM` y `plusM` es asociativo.

Programación con mónadas

La idea básica de convertir un programa a forma monádica es la siguiente: *una función de tipo* `a -> b` *es convertida en una de tipo* `a -> M b`. De este modo, la función identidad que tiene tipo `a -> a`, tendrá su correspondiente función en forma monádica en `unitM`, cuyo tipo es `a -> M a`. Lleva valores a su correspondiente representación monádica.

Las dos funciones `k :: a -> b` y `h :: b -> c` pueden ser compuestas escribiendo

(3.12)

```
\a -> let b = k a in h b
```

que tiene tipo `a -> c`. De manera similar, dos funciones en forma monádica, `k :: a -> M b` y `h :: b -> M c` se compondrán escribiendo

(3.13)

```
\a -> k a 'bindM' (\b -> h b))
```

que tiene tipo `a -> M c`. Por lo que `bindM` tiene un rol similar al de una expresión `let`. Como dijimos antes, las tres leyes para mónadas son simplemente para asegurar que esta forma de composición sea asociativa, teniendo a `unitM` como identidad a izquierda y derecha.

Tal como el tipo `Valor` representa un valor, el tipo `M Valor` puede pensarse como representado una computación. El propósito de `unitM` es forzar a un valor en una computación; el propósito de `bindM` es el de evaluar la computación, devolviendo un valor.

Informalmente, `unitM` nos mete dentro de la mónada y `bindM` nos permite movernos dentro. ¿Cómo salimos de la mónada? En general tales operaciones requieren un mayor diseño ad hoc. Para muchos propósitos, es suficiente proveer lo siguiente:

(3.14)

```
showM :: M Valor -> String
```

Cambiando las definiciones de `M, unitM, bindM,` y `showM` podemos realizar diversas mónadas que presenten una variedad de comportamientos.

3.2.3 Sintaxis especial

Hasta ahora hemos visto que no es necesario ningún tipo de soporte específico para la programación con mónadas. Sin embargo, se han propuesto dos extensiones sintácticas, actualmente en uso. Una de ellas fue mencionada antes y es una generalización de la noción de definición de listas por comprensión. La otra es utilizada sobre todo en el lenguaje Haskell y se la conoce como notación `do.`

Mónadas por comprensión

Como se describe en [39], la notación de comprensión de listas se generaliza a una mónada arbitraria, mediante la siguiente traducción:

(3.15)

```
[ t ]                    = unitM t
[ t | x <- u ]           = u 'bindM' (\x -> unitM t)
[ t | x <- u, y <- v ]   = u 'bindM' (\x -> v
                                'bindM' (\y -> unitM t)
```

Notación *do* de Haskell

En el lenguaje funcional Haskell [15] existe una sintaxis especial para el uso de mónadas. En él, el sistema de tipos Milner-Hilner ha sido extendido con un sistema de clases por lo que las operaciones `unit` y `bind` de la ahora clase `Monad` se denotan como `return` y el operador `>>=`:

(3.16)

```
class Monad m where
  return ::  a -> m a
  (>>=) ::  m a -> (a -> m b) -> m b
```

Como es muy común utilizar expresiones con la siguiente estructura:

(3.17)

```
p1 >>= \a1 ->
p2 >>= \a2 ->
...
pn >>= \an ->
f a1 a2 ... an
```

Estas expresiones tienen una lectura operacional muy natural: aplicar `p1` y llamar `a1` al valor resultante; entonces aplicar `p2` y llamar a valor resultante `a2`; ...; entonces aplicar `pn` y llamar a su valor resultante `an`; y, finalmente, combinar todos los resultados aplicando la función semántica `f`. (Aquí es muy común utilizar `return (g a1 a2 ... an)`, para alguna función g, como función semántica.)

Haskell provee una sintaxis especial para definir funciones de tal aspecto, permitiendo que sean expresadas en la siguiente forma, más atractiva:

(3.18)

```
do  a1 <- p1
    a2 <- p2
    ...
    an <- pn
    f a1 a2 ... an
```

Esta notación puede ser usada en una sola línea si se lo preferiere, haciendo uso de paréntesis y punto y comas, del siguiente modo:

(3.19)

```
do {a1 <- p1; a2 <- p2; ...; an <- pn; f a1 a2 ... an }
```

Las subexpresiones `ai <- pi` son denominadas **generadores,** pues ellos generan los valores de las variables `ai`. En el caso especial en el que no estamos interesados en el valor del generador `ai <- pi`, el generador puede ser abreviado simplemente `pi`.

3.2.4 Algunas mónadas útiles

Mónada I: la identidad

Para comenzar, definamos la mónada trivial

(3.20)

```
type  I a     = a

unitI a       = a
a 'bindI' k   = k a
showI a       = showval a
```

Esta es la denominada *mónada identidad*, `I` es la función identidad sobre tipos, `unitI` es la función identidad, `bindI` es aplicación postfija y `showI` es equivalente a `showval`.

Mónada E: manejo de errores

Para agregar mensajes de error, podemos definir la siguiente mónada.

(3.21)

```
data  E a              = Success a | Error String

unitE a                = Success a
errorE s               = Error s

(Success a) 'bindE' k  = k a
(Error s) 'bindE' l    = Error s

showE (Success a)      = "Exito:  " ++ showval a
showE (Error s)        = "Fracaso:  " ++ s
```

Cada función, por lo tanto, o bien termina normalmente devolviendo un valor de la forma `Success a`, o indica un error devolviendo un valor de la forma `Error s` donde `s` es un mensaje de error. Si `m :: E a` y `k :: a -> E` entonces `m 'bindE' k` actua como una aplicación postfija estricta: si `m` tiene éxito entonces se aplica `k` al resultado exitoso; si `m` falla entonces de igual modo lo hace la aplicación. La función `show` muestra o bien el resultado exitoso o el mensaje de error.

Mónada S: Estado

La mónada de los transformadores de estado, se define como sigue.

(3.22)

```
type  S a     = State -> (a, State)

unitS a       = \s0 -> (a, s0)
m 'bindS' k   = \s0 -> let (a,s1) = m s0
                           (b,s2) = k a s1
                       in (b,s2)
```

Un transformador de estado toma un estado inicial y devuelve un par formado por un valor y un nuevo estado. La función `unit` devuelve el valor dado y propaga el estado sin cambiarlo. La función `bind` toma un transformador de estado `m :: S a` y una función `k :: a -> S b`. Le pasa el estado inicial al transformador `m`, esto devuelve un valor junto con un estado intermedio; la función `k` se aplica al valor, devolviendo un transformador de estado (`k a :: S b`), al cual se le pasa el estado intermedio; este devuelve el par con el resultado y el estado final.

Dos funciones útiles en esta mónada son:

(3.23)

```
fetch       ::  S State
fetch       = \s -> (s,s)

assign      ::  State -> S ()
assign s'   = \s -> ((),s')

init        ::  State -> S x -> x
init s x    = let (x',s') = x s in x'
```

La primera devuelve el estado actual, dejándolo sin cambios. La segunda descarta el viejo estado, asignando al nuevo estado a un valor dado. Aquí `()` es el tipo que tiene solamente al valor `()`. La tercera función opera el transformador de estado `x` a un determinado estado inicial `s`; devuelve el valor computado por el transformador de estado, descartando el estado final.

Mónada L: elección no-determinista

Las elecciones no-deterministas pueden modelarse a través de la utilización de una lista de resultados.

La mónada de las listas se define del siguiente modo.

(3.24)

```
type  L a     = [ a ]

unitL a       = [ a ]
m 'bindL' k   = [ b | a <- m, b <- k a ]

zeroL         = []
l 'plusL' m   = l ++ m

showL m       = showlist [ showval a | a <- m ]
```

Las definiciones anteriores están expresadas en la notación usual de comprensión de listas. La función `showlist` lleva una lista de cadenas en una cadena, con la puntuación adecuada.

Al poder definir las funciones `zeroL` y `plusL` vemos que las listas son un caso de mónadas con un cero y un más (véase defnción (3.10)). En este caso, es de notar que la existencia de tales funciones nos permite utilizar filtros en la notación de comprensión de mónadas.

3.2.5 Analizadores sintácticos

En la siguiente sección desarrollaremos una mónada de *parsing* (análisis sintáctico) y diversos combinadores útiles. Nos centraremos principalmente en [16] y en el lenguaje de programación Haskell.

El tipo de los *parsers*

Comencemos pensando a un *parser* como una función que toma una cadena de caracteres como entrada y devuelve algún tipo de árbol como resultado, con la intención de que dicho árbol haga explícita la estructura gramatical de la cadena:

(3.25)

```
newtype Parser = String -> Tree
```

En general, sin embargo, un *parser* puede no consumir toda su cadena de entrada, asi que el resultado en vez de ser tan sólo un árbol, debemos también devolver el sufijo no consumido de la cadena de entrada. Por lo que modificamos nuestro tipo de *parsers* del siguiente modo:

(3.26)

```
newtype Parser = String -> (Tree, String)
```

Similarmente, un *parser* puede fallar sobre su cadena de entrada. En vez de devolver un error en tiempo de ejecución si tal cosa sucediera, podemos elegir que los *parsers* devuelvan una lista de pares en vez de un solo par, con la convención que la lista vacía denota el fracaso del *parser* y la lista de un solo elemento denota éxito:

(3.27)

```
newtype Parser = String -> [(Tree, String)]
```

Tener una representación explícita del fracaso y devolver la parte de la cadena de entrada no consumida hace posible definir combinadores para construir *parsers* a partir de *parsers* más pequeños como si fuesen piezas. Devolver una lista de resultados abre la posibilidad de devolver más de un resultado si la cadena de entrada puede ser analizada en más de una forma, lo cual puede ser el caso si la gramática subyacente es ambigua.

Finalmente, diferentes *parsers* devolveran seguramente diferentes tipos de árboles, por lo que es útil abstraernos del tipo específico `Tree` de los árboles, y tomar al tipo de los valores resultantes como un parámetro del tipo `Parser`:

(3.28)

```
newtype Parser a = String -> [(a, String)]
```

Una mónada para *parsers*

El primer *parser* que definimos es `item`, el cual consume exitosamente el primer caracter si su cadena de argumento es no vacía, y falla en caso contrario:

(3.29)

```
item :: Parser Char
item = Parser (\cs -> case cs of
                            ‘‘’’     -> []
                            (c:cs) -> [(c,cs)])
```

Paso seguido definimos dos combinadores que reflejan la naturaleza monádica de los *parsers.* Para ello utilizamos la clase predefinida `Monad` del lenguaje Haskell (véase (3.16)), conviertiendo al constructor de tipos `Parser` en una instancia de dicha clase:

(3.30)

```
instance Monad Parser where
  return a = Parser (\cs -> [(a,cs)])
  p >>= f = Parser (\cs -> concat [parse (f a) cs’ |
               (a,cs’) <- parse p cs])
```

El *parser* `return a` termina exitosamente sin consumir ningún elemento de su parámetro cadena, y devuelve el valor `a` solo. El operador `(>>=)` es un operador de *secuenciamiento* de *parsers.* Usando una función desconstructora para *parsers* definida por `parse (Parser p) = p`, obtengo los resultados de la forma `(a,cs’)`, donde `a` es un valor y `cs’` una cdena. Para cada par de tal forma, `f` es una *parser* que se aplicará a la cadena `cs’`. El resultado es una lista de listas, la cual es concatenada para obtener la lista final de resultados.

Las funciones `return` y `(>>=)` sobre *parsers* satisfacen las leyes 3.2.1 discutidas antes. Las leyes de identidad permiten hacer más sencillos algunos *parsers,* y la ley de asociatividad permite eliminar los paréntesis en los secuenciamientos repetidos.

Aquí nos es de gran utilidad la notación *do* del Haskell analizada en 3.2.3. De este modo, un *parser* que consuma tres caracteres, descarte el segundo caracter y devuelva los otros dos en un par, puede ser definido del siguiente modo:

(3.31)

```
p ::  Parser (Char,Char)
p = do {c <- item; item; d <- item; return (c,d)}
```

Combinadores de elección

En Haskell la noción de mónada con un cero está capturada por la clase predefinida `MonadZero` así como la de mónada con un más, por la clase `MonadPlus` (véase (3.10)):

(3.32)

```
class Monad m => MonadZero m where
  zero ::  m a

class Monad m => MonadPlus m where
  (++) ::  m a -> m a -> m a
```

El constructor de tipos `Parser` puede convertirse en una instancia de tales clases del siguiente modo:

(3.33)

```
instance MonadZero Parser where
  zero   = Parser (\cs -> [])

instance MonadPlus Parser where
  p ++ q = Parser (\cs -> parse p cs ++ parse q cs)
```

El *parser* `zero` falla para toda cadena de entrada, no devolviendo nada. El operador `(++)` es un operador de *elección* (no determinista) para los *parsers*. El *parser* `p ++ q` aplica ambos *parsers* `p` y `q` a la cadena de entrada, y une su lista de resultados.

Es fácil ver que con estas definiciones de `zeroParser` y `plusParser`, `Parser` verifica las leyes (3.11). Más aun, para el caso especial de los *parsers*, también puede demostrarse quie —módulo el ligamiento que involucra `(>>=)`— `zero` es un cero a izquierda y a derecha de `(>>=)`, y que `(>>=)` distribuye a la derecha respecto de `(++)`, y que (si ignoramos el orden de los resultados devueltos por los *parsers*) `(>>=)` distribuye a la izquierda respecto de `(++)`:

(3.34)

```
              zero >>= f   = zero
        p >>= const zero   = zero
          (p ++ q) >>= f   = (p >>= f) ++ (q >>= f)
p >>= (\a -> f a ++ g a)   = (p >>= f) ++ (p >>= q)
```

La ley del cero permite hacer más sencillos algunos *parsers*, mientras que las leyes distributivas permiten mejorar la eficiencia de algunos otros *parsers*.

Los *parsers* construidos con `(++)` devolveran muchos resultados si la cadena de entrada puede ser analizada de distintos modos. En la práctica, estamos normalmente interesados en el primer resultado. Por esta razón, definimos el operador de elección (determinista) `(+++)` con el mismo comportamiento de `(++)`, excepto que devolverá a lo más un resultado:

(3.35)

```
(+++) ::  Parser a -> Parser a -> Parser a
p +++ q = Parser(\cs->case parse (p ++ q) cs of
                        []       -> []
                        (x:xs) -> [x])
```

Todas las leyes expresadas antes para `(++)` valdrán también para `(+++)`. Sin embargo, para el caso de `(+++)`, se satisface automáticamente la precondición para la ley distributiva a izquierda.

Otros combinadores

El *parser* `item` consume caracteres incondicionalmente. Para permitir anaálisis condicional, definimos el combinador `sat` que toma como entrada un predicado, y devuelve un *parser* que consume un solo caracter si éste satisface el predicado, fallando en caso contrario:

(3.36)

```
sat ::  (Char -> Bool) -> Parser Char
sat p = do  c <- item; if p c then return c else zero
```

Con él podemos definir un *parser* para un caracter específico:

(3.37)

```
char ::  Char -> Parser Char
char c = sat (c ==)
```

Y, gracias a él, un *parser* para una cadena específica:

(3.38)

```
string          ::  String -> Parser String
string ‘‘’’     = ‘‘’’
string (c:cs)   = do char c; string cs; return (c:cs)
```

En general, cualquier gramática libre de contexto puede ser analizada utilizando los combinadores de concatenación y elección.

Capítulo 4

El código

Siguiendo los lineamientos de los capítulos anteriores se realizo un analizador sintactico en el lenguaje Haskell utilizando los siguientes modulos:

`LFGmain.hs` Función `main` que conecta entre sí los diversos módulos y se encarga de la entrada salida.

`LFGgram.hs` Gramática de los archivos de datos de ingreso (gramática LFG, entradas léxicas y restricciones).

`LFGpp.hs` *Pretty printing* de las distintas estructuras de datos.

`LFGtipos.hs` Estructuras de datos utilizadas.

`LFG1.hs` Función `parser`, la función principal del código.

El programa final recibe por línea de comandos la oración a analizar y abre tres archivos predeterminados con los datos de la gramática, el léxico y las restricciones. Su salida es una versión *p.p.* de la lista de estructuras-f que la gramática asocia a dicha oración. La lista puede estar vacía (la gramática predice que la oración ingresada no pertenece al castellano) o tener más de un elemento (la oración es ambigua para esta gramática).

4.1 Tipos de datos

En `LFGtipos.hs` se definen los tipos `GramF`, `LexicoF`, `RestrF`, `ArbolF`, `EstrF` y sus subtipos. Analicemos uno a uno dichos tipos.

4.1.1 `GramF`

Este tipo alberga una representación de una gramática LFG. Consiste en una lista de reglas (`ReglaF`).

(4.1)

```
type GramF = [ReglaF]
```

Cada regla es una upla que consta de un antecedente (primera componente) y un consecuente (segunda componente). En el caso de reglas gramaticales normales, el antecedente seria un *string* de símbolos y el consecuente una lista de *strings*. en el caso de las gramáticas léxico funcionales, como ya hemos visto en la subsección 2.1.1, se agregan también los esquemas funcionales para cada *string* del consecuente. Asi pues, las reglas serán:

(4.2)

```
type ReglaF = (String, [(String,[EsqF])])
```

Un esquema funcional es una ecuacion donde se encuentran igualadas dos expresiones que involucran estructuras-f. Implementamos esto como un par de tipos reprsentando dichas expresiones

(4.3)

```
type EsqF = (ExprF, ExprF)
```

Una expresion involucrando estructuras-f será, o bien un valor dado (3, $SING$) o bien una expresión que represente una estructura-f

(4.4)

```
data ExprF   = EF ExprEstrF
             | EFvalor ValorF
```

Las metavariables *padre* y *misma* son las primeras candidatas como representantes de estructuras-f. Hay que agregar las construcciones de la forma f_5 a las que denominaremos “punteros a estructuras-f” o simplemente, “punteros-f”. Estos son nuestras formas primitivas, las formas complejas pueden construirse a partir de acceder a líneas de la estructura-f. (Otra forma de ver esto es pensar que estamos aplicando una función a la estructura-f).

(4.5)

```
data ExprEstrF  = EEFPadre
                | EEFMisma
                | EEFPuntf Int
                | EEFapl ExprEstrF FuncionF
```

Mientras tanto, los valores que mencionábamos antes pueden ser de distintos tipos, predicados (como en ‘JUGAR $<(f_5\ SUJ)(f_5\ OBJ)>$’), sintagmas (S), estructuras-f, punteros a estructuras-f, enteros, y posiblemente otros valores (la representación es lo suficientemente abierta como para permitir incorporar fácilmente nuevos tipos de valores.

(4.6)

```
data ValorF  = VFpred Pred
             | VFnum Int
             | VFsint Sintagma
             | VFestrf EstrF
             | VFpunt Int
```

Un predicado será, siguiendo los ejemplos vistos, una semántica con posibles argumentos.

(4.7)

```
type Pred = (Semantica, [Arg])
```

La `Semantica` actualmente esta implementada como un string pero puede ser un tipo de datos más complejo e inclusive incluir funciones que, en el caso de los verbos, nos den el “efecto” de la oracion sobre sus argumentos (por ejemplo, SUJ y OBJ). Los argumentos son una lista de sintagmas aunque deberían ser secuencias de `EsqF` modificación que será realizada en el futuro.

4.1.2 `LexicoF`

Un lexico en una gramática léxico funcional es un conjunto de entradas léxicas, según vimos en la subsección 2.1.2.

(4.8)

```
type LexicoF = [EntrL]
```

Un entrada léxica, contiene (figura 2.2) la palabra a quien se refiere, la clase de palabra a la que pertenece (V, A, S, ADV, etcétera) y un conjunto de esquemas funcionales que son utilizados al incluir la entrada en el árbol.

(4.9)

```
type EntrL = (String, Sintagma, [EsqF])
```

Este tipo fue dotado de funciones de acceso y proyectores para facilitar cualquier tipo de cambios:

(4.10)

```
elGetP (p, s, l) = p
elPutP p' (p, s, l) = (p', s, l)
elGetS (p, s, l) = s
elPutS s' (p, s, l) = (p, s', l)
elGetL (p, s, l) = l
elPutL l' (p, s, l) = (p, s, l')
elVacia = ('','',[])
```

4.1.3 `ArbolF`

El resultado del *parser* será un árbol con las anotaciones funcionales. Como implementación de dicho árbol utilizamos el siguiente tipo, el cual es un árbol n-ario con entradas léxicas en las hojas.

(4.11)

```
data ArbolF  = AFhoja EntrL
             | AFnodo NodoF [ArbolF]
```

El tipo central aquí es el del nodo en cada árbol, al que denominamos `NodoF`. En él encontramos el sintagma asociado al nodo, los esquemas funcionales y el entero con el que luego se numeran los nodos.

(4.12)

```
type NodoF = (Sintagma, [EsqF], Int)
```

Este tipo también ha sido dotado de funciones de acceso para facilitar su modificación:

(4.13)

```
nfGetS (s, _, _) = s
nfPutS s' (s, e, n) = (s', e, n)
nfGetE (_, e, _) = e
nfPutE e' (s, e, n) = (s, e', n)
nfGetN (_, _, n) = n
nfPutN n' (s, e, n) = (s, e, n')
nfVacio = ('',[],0)
```

4.1.4 `EstrF`

El tipo de las estructuras-f es central a todo el sistema. Siguiendo el tratamiento relativo a las estructuras-f (sección 2.5), las consideramos como una secuencia de líneas.

(4.14)

```
type EstrF = [LineaF]
```

Cada línea es un par atributo–valor. Los valores son los `ValorF` ya definidos en (4.6).

(4.15)

```
type LineaF = (String, ValorF)
```

4.1.5 Otros tipos

Un tipo importante es el utilizado para representar a las descripciones funcionales (sección 2.4).

(4.16)

```
type DescrF = [EsqF]
```

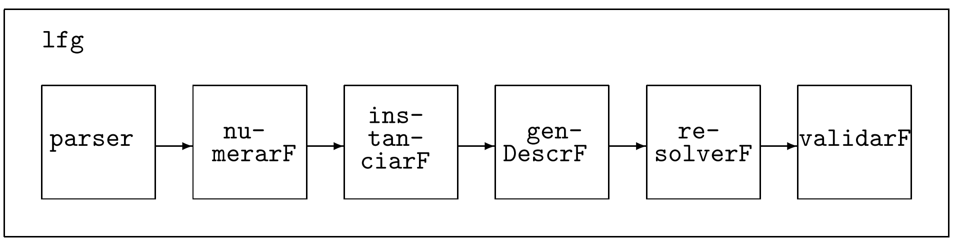


Figura 4.1: Descripción de la función principal (función `lfg`).

También hemos mencionado en varios otros puntos el tipo `Sintagma`. Actualmente está implementado como una *string* pues es la implementación más flexible. Si el problema fuese bien fijo y delimitado, sería más eficiente implementarlo como una enumeración (i.e., fijar la cantidad de sintagmas a un número fijo y conocido).

(4.17)

```
type Sintagma = String
```

4.2 Función principal

La función principal (función `lfg`) recibe como entrada el texto a analizar, la gramática y el léxico, devolviendo una lista de estructuras-f como resultado. Esta función realiza los pasos explicados en el capítulo 2: primero genera un árbol a partir de la oración, la gramática y el léxico (función `parser`), dicho árbol luego es numerado, esto es, se asigna un número único a cada nodo, representando la estructura-f asociada a él (función `numerarF`). Con los nodos numerados, ahora podemos darle sentido a las metavariables `EEFPadre` y `EEFMisma`, reemplazandolas por un `(EEFPunt` *i*`)` al nodo que corresponda (función `instanciarF`). De este modo las ecuaciones funcionales ya están formadas, se debe recorre el árbol extrayéndolas; de este modo se obtiene la descripción funcional (función `genDescrF`). Este sistema de ecuaciones simbólicas es resuelto en la función `resolverF`. Finalmente, los criterios de completitud y corrección de las soluciones generadas se verifican en la función `validarF`. Una representación gráfica podemos verla en la figura 4.1.

4.2.1 `parser`

Esta función interpreta la gramática utilizando los combinadores mónadicos de análisis sintáctico. La función más importante es `interpParser` que interpreta la gramática para un símbolo dado. La función `parser` se reduce a hacer `interpParser` para el símbolo representando todo la oracion, O, sobre el *string* de entrada.

La tarea de interpretación se divide en dos partes: si el símbolo no terminal en cuestión tiene entradas léxicas asociadas (por ejemplo, SN, un sintagma nominal, no tiene entradas léxicas asociadas, sólo producciones; S, sustantivo, en cambio, sí las tiene), se verifica si es posible reconocer alguna de tales entradas. Esta es la "parte léxica" de la interpretación. Aparte de ella, se realiza la unión (respecto de la mónada `Parser`, como se vió en la subsección 3.2.5) con el *parser* que resulta de interpretar todas las producciones que en su lado izquierdo tienen al símbolo que se está interpretando. La interpretación de una producción la realiza la función local `ejecutarProd`, la cual secuencia los parsers obtenidos de interpretar en orden cada símbolo de la parte derecha de la producción dada.

Eficiencia

Evidentemente esta solución no es la más eficiente. En primer lugar es interpretada. En [24] se describen métodos más eficientes basados en compilación. Sin embargo, la interpretación tiene la ventaja de su sencillez para cambiar la gramática de forma dinámica, haciendo pruebas y modificaciones. Para tener una gramática "en producción" el enfoque compilado es ideal. Otra forma en que se vería incrementada la eficiencia del algoritmo sería a través de técnicas de **memoización**. En este caso puntual, habría que introducir a `interParser` dentro de una mónada de estado que sirva a manera de memoria de los análisis ya realizados. De este modo se puede lidiar con el recomputo en el estudio de ciertas ramas del árbol de solución de datos ya obtenidos en otras ramas, permitiendo asi bajar el orden del algoritmo.

Respecto del léxico, la solución actual tiene dos restricciones: una de tamaño pues no se escala en lo absoluto (el reconocimiento de un sustantivo anda bien si son 30 pero no 3,000) y, sobre todo, no se adapta a frases completas, muy comunes en el castellano, de estilo de "a pesar de" y similares. Actualmente esta falencia podría solucionarse con un preprocesado de la oración que reemplace este tipo de construcciones con "palabras simuladas" del estilo *a_pesar_de*, las cuales sí pueden figurar en el diccionario. De cualquier modo la mejor solución es alimentar al intérprete de la gramática con una secuencia ahora ya no de palabras (caracteres que forman palabras, en realidad) sino de una lista de listas de entradas léxicas, donde cada elemento significa todas las entradas léxicas que pudieron ser asociadas a una determinada palabra, en el orden correspondiente.

Construcción del árbol

La función `interpParser` recibe, además, como parámetro el esquema funcional con el que la regla-f (por la cual se lo está analizando) lo asociaba. Por ejemplo, si tenemos una regla $SV \rightarrow V(\uparrow=\downarrow)SN(\uparrow OBJ =\downarrow)$, cuando se interprete el símbolo SN a `interpParser` se le pasará como parámetro $(\uparrow OBJ =\downarrow)$. Con esta información `interpParser` combina el resultado de los subparsers y vía `map` devuelve un árbol n-ario de tipo `arbolF`.

Ambigüedad

La función `interpParser` está dentro de una mónada `Parser` que trabaja con una lista de todos los posibles *parsings* que se están generando. La función `parser`, en cambio, devuelve una lista de todos los `arbolF` de los *parsings* que consumieron todo el *string* de entrada (esto es, todo los análisis que resultaron exitosos). Todos estos árboles son tenidos en cuenta en los análisis posteriores. Se espera que `resolverF` y `validarF` puedan deshacerse de los *parsings* erróneos o superfluos pero muchas oraciones o bien son ambiguas también para un hablante nativo o bien necesitan de información semántica para desambiguarse.

4.2.2 `numerarF, instanciarF` y `genDescrF`

Estas funciones son bastante sencillas. Las tres recorren el árbol realizando sus tareas específicas.

`numerarF`

Esta función utiliza una mónada de estado `MST` donde el estado es el próximo número a otorgar. La función `numerarF`, entonces, extrae desde la mónada el nuevo árbol calculado por `numerarFM`, función que recorre el árbol llenando los campos en los `NodoF` (véase (4.12)), utilizando la función de acceso `nfPutN` como se explicó en (4.13). Para el paso recursivo se utiliza la función auxiliar `numerarListaFM` que realiza lo propio sobre una lista de subárboles.

`instanciarF`

Usando una función auxiliar `instanciarF'` que reciba como parámetro el número asociado al nodo padre, `instanciarF` recorre el árbol reemplazando en los esquemas funcionales las metavariables `EEFPadre` por dicho parámetro y `EEFMisma` por el número asociado al nodo actual.

Aquí es válida una aclaración: este esquema no está soportando construcciones que permitan acceder al "abuelo" de un nodo u otros ancestros, del estilo ($\uparrow\uparrow$ SUJ) pero esto no es inconveniente pues parece que tales expresiones no aparecen en LFG.

`genDescrF`

Esta función simplemente recorre el árbol quedándose con los esquemas funcionales asociados a cada nodo, los cuales conatena devolviéndolos en una lista.

4.2.3 `resolverF`

La resolución del sistema de ecuaciónes simbólicas es, sin duda, la pieza de software más compleja del analizador sintáctico. Se utilizó en su construcción una formalización del método descripto en el capítulo 2, a partir de 2.4.

El algoritmo se puede resumir del siguiente modo:

1. Se llevan todas las ecuaciones a tres formas canónicas:

   (4.18)

   $$f_i = f_j \quad (f_i\ A) = B \quad (f_i\ A) = f_j$$

   De acuerdo a lo hablado antes respecto de `instanciarF` las otras ecuaciones que pueden aparecer son formas más complejas del estilo de

   (4.19)

   $$((f_i\ A)\ B) = C \quad \text{o} \quad ((f_i\ A)\ B) = f_j$$

   que pueden ser llevadas a la forma canónica mediante la introducción de nuevas variables de estructura-f auxiliares, es decir, haciendo:

   (4.20)

   $$((f_i\ A)\ B) = C \qquad \rightarrow \qquad \begin{array}{l} (f_i\ A) = f^* \\ (f^* B) = C \end{array}$$

Esta tarea la realiza la función `limpiarDescrF` que toma una `DescrF` y devuelve otra "limpia".

2. En las ecuaciones de la forma canónica en (4.18) distinguimos dos tipos, las de la forma $(f_i\ A) = B$ y $(f_i\ A) = f_j$ de tipo *constructivo* y las de la forma $f_i = f_j$, de tipo *adhesivo.* Las ecuaciones generadas en el paso anterior son ahora clasificadas (función `clasificarEcF`) en alguno de los dos tipos.

3. Se crea la estructura de datos de la `TablaF` que se explicará más abajo. Dicha estructura alberga todas las estructuras-f en proceso de construcción. En la función `resolverF1` se resuelven las ecuaciones de tipo constructivo, agregando líneas a la estructura-f siempre que sea posible (puede no ser posible si la estructura-f ya tiene una línea con esa clave y un dato distinto). En dicho caso de imposibilidad, estamos frente a una demostración de que el conjunto de ecuaciones funcionales es inconsistente, con lo que se procede a abortar la resolución (el proceso de salida con error es facilitado teniendo a estas funciones dentro de una mónada `Maybe`, versión Haskell de la mónada E analizada en 3.2.4).

4. Las ecuaciones de tipo adhesivo son tenidas en cuenta en la función `resolverF2`, que procede a igualar mediante el algoritmo de **mezcla** visto en 2.5.3. La implementación de dicho algoritmo sobre la tabla-f es un punto donde se debe tener cuidado, véase más adelante. Dicha implementación la encontramos en la función `mezcla`.

5. Finalmente se extrae de la tabla-f la estructura correspondiente a toda la oración.

`TablaF`

Este tipo de datos es la versión en funcional de un arreglo de punteros a estructuras-f en imperativo. Como punteros utilizamos **identificadores de estructuras-f**, esto es, números enteros a los que damos el tipo `IdEF`. El tipo de las estructuras-f es ampliado para contener la lista de identificadores por la que se la conoce, en el tipo `UEstrF`. Una `TablaF` no será, entonces, más que una lista dichas estructuras-f extendidas. Un detalle importante: como vimos en (4.6), `ValorF` admite dos formas de contener una estructura-f: una es teniendo ciertamente a la estructura-f allí, mediante el constructor `VFestrf`, la otra es indicando una *referencia* a ella, con el constructor `VFpunt`. Ahora bien, todas las estructuras-f que aparecen en `TablaF`, por la forma en que se la va construyendo, son tales que en ninguna `LineaF` figura un `ValorF` construido con `VFestrf`, sin embargo, estas estructuras-f si tienen valores anidados, a través de los constructores `VFpunt`. Es por eso que en el paso 5 del algoritmo la estructura-f final debe ser **extraída de la tabla**, la información relativa a ella está presente, pero debe calcularse.

1,5,6	$\begin{bmatrix} ... & ... \\ PRED & \text{`JUAN'} \\ NUM & SING \\ PERS & 3 \\ ... & ... \end{bmatrix}$
2,3	$\begin{bmatrix} ... & ... \\ A & B \\ OBJ & \texttt{VFpunt} \\ ... & ... \end{bmatrix}$
...	...

Figura 4.2: Tabla-f.

Una representación pictórica de la tabla puede verse en 4.2

4.2.4 `validarF`

La última función de la figura 4.1 se encarga de los asuntos descriptos en la sección 2.6.2, completitud y coherencia. La funcion recibe como parámetros la estructura-f a validar y un listado de funciones gramaticales (i.e., *strings*) que anteriormente una función auxiliar, `genArg`, extrajo del léxico. Estas funciones gramaticales son las que aparecen subcategorizadas en formas semánticas, y son importantes para el chequeo de coherencia. `validarF` devuelve un valor booleano indicando si la estructura-f es completa y coherente o falla en alguna de estas dos propiedades. El chequeo de completitud consiste en verificar la existencia de todas los argumentos de las formas semánticas. Para el chequeo de coherencia se hace uso del listado de argumentos antes mencionado y se verifica que no exista en la estructura-f ningun argumento de dicha lista no pedido por alguna forma semantica existente. Es decir, si en alguna entrada léxica figura que OBJ es un argumento semántico, entonces una estructura-f dada en la que OBJ este presente pero no figure entre los argumentos pedidos en el $PRED$ correspondiente será inválida (este podría ser el caso, por ejemplo, de una oración mal formada en la cual a un verbo intransitivo se le está asociando un objeto directo).

4.3 Otros módulos y funciones

4.3.1 `LFGgram`

Este módulo contiene todos los elementos necesarios para levantar archivos de texto conteniendo gramáticas y léxicos en formatos dados. Hace fuerte uso de la librería de *parsing* mónadico descripta sucintamente en 3.2.5.

Las funciones principales aquí son

(4.21)

```
lfgGram    ::  String -> GramF
lfgLexico  ::  String -> LexicoF
```

El formato de los archivo es *free form*, esto es, los datos pueden disponerse de cualquier forma y en varias líneas, sin hacer distinción entre los saltos de línea, los espacios y las tabulaciones.

He aquí la gramática libre de contexto que describe a los archivos de las gramáticas:

(4.22)

```
Gram       →  (ReglaF ';')*
ReglaF     →  Ident '->' SintEsqF*
Ident      →  alphanum+
SintEsqF   →  Ident '(' (EsqF ';')* ')'
EsqF       →  ExprF '=' ExprF
ExprF      →  ExprExstrF | ValorF
ExprEstrF  →  Padre | Misma | Apl
Padre      →  'up'
Misma      →  'dn'
Apl        →  '(' ExprEstrF Ident ')'
ValorF     →  Pred | DeInt | DeSint
Pred       →  ''' Ident '<' Ident* '>' '''
DeInt      →  integer
DeSint     →  Ident
```

Basándones en la gramática anterior, agregando las siguientes dos producciones obtenemos la gramática libre de contexto de los archivos de los léxicos:

(4.23)

```
Lex     →  (EntrL ';')*
EntrL   →  Ident Ident '(' (EsqF ',')* ')'
```

4.3.2 `LFGpp`

Este módulo tiene un conjunto de funciones que sirven para realizar el *pretty printing* de los distintos tipos de datos difinidos en `LFGtipos.hs`. Inicialmente la salida no se correspondia exactamente con el formato aceptado por `LFGgram` pero para la implementación de la interface esto fue necesario ampliándose las funciones existentes a versiones con la terminación `EnLin` para indicar que generan todo el tipo de datos en una sola línea de texto. Las funciones principales aquí son `ppGramF`, `ppLexicoF`, junto con sus contrapartes `ppGramEnLin` y `ppLexEnLin`.

Un detalle que debería agregarse a este módulo es la utilización de estas rutinas de *p.p.* como implementaciones del método `show` lo que permitirá a los tipos de datos de `LFGtipos` ser considerados como pertenecientes a la clase `Show`, lo cual es muy útil a la hora de realizar la depuración de los programas y para generar mensajes de error más comprensibles.

4.4 Interface gráfica

4.4.1 Generalidades

Para permitir la utilización del analizador de manera práctica, eficiente y cómoda se le adiciona una interface vía WWW como en [7] (aunque nuestra aproximación es distinta a la del `getarun` pues permitimos analizar oraciones *on the fly*, cambiar la gramática y el léxico, siendo nuestro paradigma funcional compilado y no prolog interpretado).

Además, para difundir el uso del programa se pensó en una política de **cuentas**, cada una con sus propias gramáticas y léxico asociados.

De igual modo, la búsqueda de gramáticas que caractericen determinadas particularidades de un idioma es una actividad experimental en la cual muchas veces, por prueba y error se va aproximando a un modelo más acabado. Por ello, entonces, la interface debe ser ágil y permitir probar varias hipótesis sintácticas simultáneamente.

Web y Haskell

La WWW y en particular la programación de sitios Web o **CGI**s como se los denomina está en gran medida copada por la utilización del lenguaje PERL (un lenguaje imperativo interpretado, especializado en la generación de reportes, que permite la utilización de complejas *patterns* y diversas operaciones sobre *strings*). Sin embargo los lenguajes funcionales son una mejor opción, pues son más fáciles de programar y la velocidad de la red (donde se encuentra en general el cuello de botella) lima pequeñas diferencias de velocidad entre imperativo y funcional.

Sobre esta líneas Erik Meijer [26] desarrolló una librería de módulos haskell para escribir CGIs. Esta librería se distribuye conjuntamente con el intérprete **Hugs**. En ella se plantea al intercambio CGI en un muy alto grado de abstracción y se incluye una clase "envolvedora" (`Wrapper.hs`) que oculta la mayor parte de los "detalles oscuros" de la programación CGI.

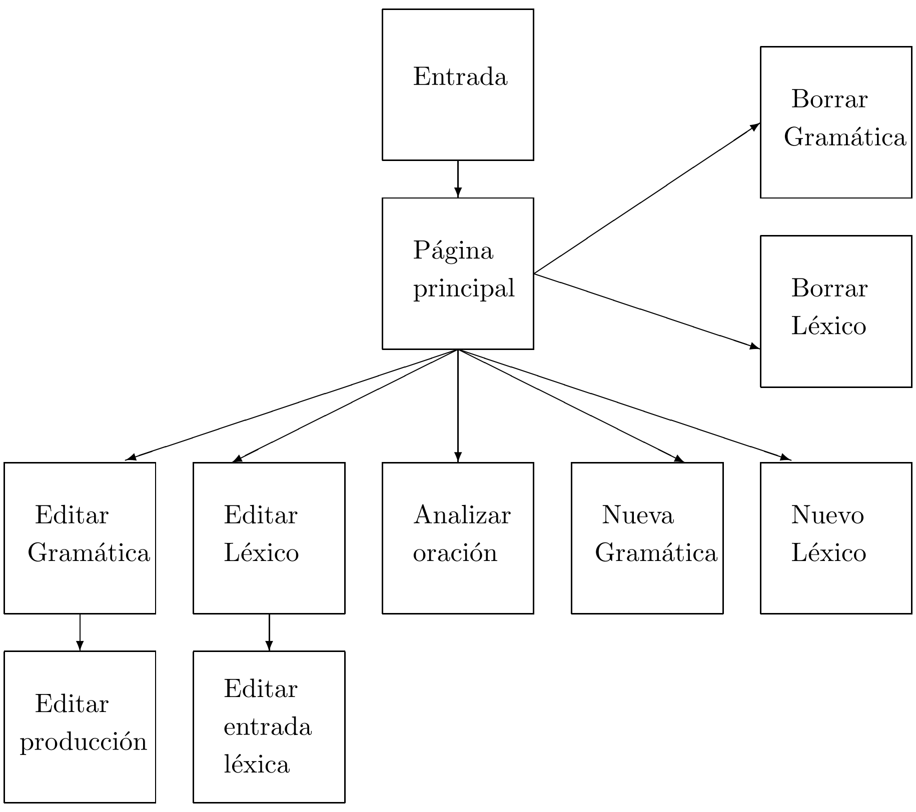


Figura 4.3: Árbol del sitio Web que forma la interface.

En la librería se encuentran, además, una poderosa abstracción del lenguaje HTML como tipo de datos haskell (`HTML.hs`) y funciones que asisten en la creación de páginas (`HTMLWizard.hs`).

The Glorious Glasgow Haskell Compilation System

Inicialmente se utilizaba Hugs 1.4 (*Haskell User's Gofer System*) para soportar todo el sitio (a travéz de su interface como lenguaje de *scripting*, `runhugs`) pero pronto la eficiencia por un lado y la falta de rutinas complejas de manejo de archivos (por ejemplo, el borrado de archivos) por el otro determinó la necesidad de cambiar a un enfoque compilado, para el que se optó por el GHC. Se utilizó la versión 2.10 de dicho compilador, una versión estabilizada y en funcionamiento (pruebas anteriores con la versión 2.05 no fueron tan positivas). Con esta versión se utilizarón, además, las extensiones POSIX al lenguaje Haskell (en particular, aquellas que permiten cambiar los atributos a un archivo).

4.4.2 Descripción interna

El sitio consta de 11 páginas de Web, dispuestas formando el árbol de la figura 4.3.

La interface consta de los siguientes archivos:

`LFG.hs`	Función `main` que ejecuta el LFG_cgi correspondiente, según el nombre del ejecutable.
`LFG_cgi1.hs`	Manejador CGI página de entrada.
`LFG_cgi2.hs`	Manejador CGI página principal.
`LFG_cgi211.hs`	Manejador CGI editar gramática.
`LFG_cgi2111.hs`	Manejador CGI editar producción.
`LFG_cgi212.hs`	Manejador CGI editar léxico.
`LFG_cgi2121.hs`	Manejador CGI editar entrada léxica.
`LFG_cgi221.hs`	Manejador CGI nueva gramática.
`LFG_cgi222.hs`	Manejador CGI nuevo léxico.
`LFG_cgi23.hs`	Manejador CGI analizar oración.
`LFG_cgi241.hs`	Manejador CGI borrar gramática.
`LFG_cgi242.hs`	Manejador CGI borrar léxico.
`LFG_pages.hs`	Módulo concentrador información sobre páginas.
`LFG_pagesc.hs`	Constantes relacionadas con la dirección del servidor, nombres de las páginas, etcétera.
`LFG_pages1.hs`	Construye la página de entrada.
`LFG_pages2.hs`	Construye la página principal.
`LFG_pages3.hs`	Construye la página de edición de la gramática.
`LFG_pages4.hs`	Construye la página de edición de una producción.
`LFG_pages5.hs`	Construye la página de edición del léxico.
`LFG_pages6.hs`	Construye la página de edición de una entrada léxica.
`LFG_pages7.hs`	Construye la página de creación de gramáticas.
`LFG_pages8.hs`	Construye la página de creación de léxicos.

`LFG_pages9.hs`	Construye la página de análisis de oraciones.
`LFG_pages10.hs`	Construye la página de borrado de gramáticas.
`LFG_pages11.hs`	Construye la página de borrado de léxicos.
`LFG_cgiLib.hs`	Funciones generales utilizadas en las rutinas CGI del sistema.
`HTMLpad.hs`	Extensión al HTMLWizard de desarrollo propio con manejo mejorado de tablas (incluyendo la función `table3D` utilizada en todo el sistema).

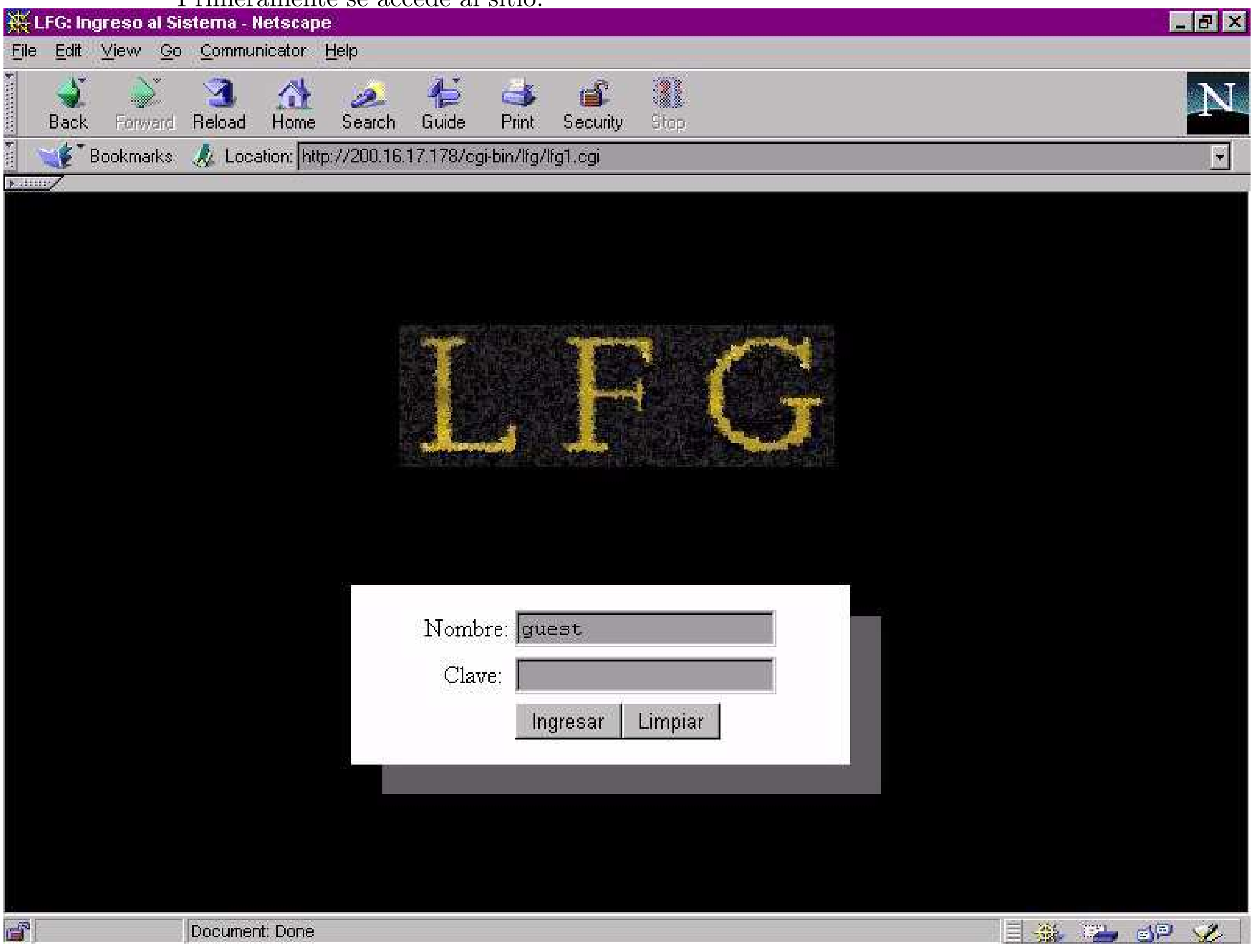
Se utilizó el siguiente esquema: cada página tiene un módulo asociado que la genera. Sin embargo, la función que realiza esta tarea no está dentro de la mónada `IO`, esto es, no puede realizar entrada/salida por lo que recibe como parámetros todos los datos que puede llegar a necesitar.

Aparte, cada página tiene asociado el programa CGI que la maneja. Para simplificar el desarrollo, distribución, etcétera, el sistema consiste en un solo ejecutable que luego debe ser enlazado (mediante links estáticos UNIX) a cada uno de los CGIs correspondientes. El módulo principal (`LFG.hs`), entonces, analiza el nombre del ejecutable con el cual se lo llamó y deriva de ahí al `main` correspondiente.

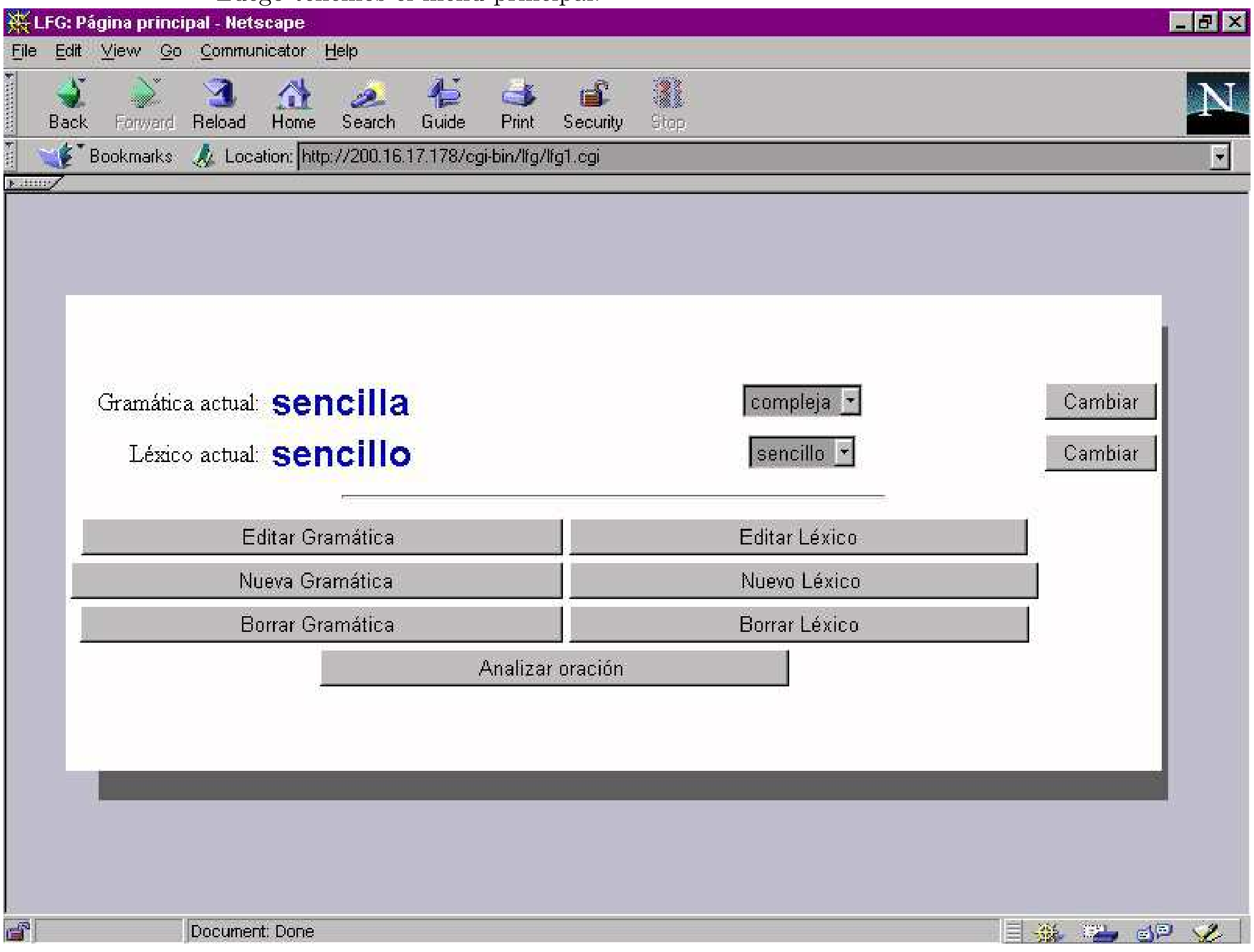
Dentro del manejador CGI se analizan las variables devueltas por la aplicación, se abren y procesan los archivos que sean necesarios y se devuelve una nueva página (pero no necesariamente la página que administra dicho manejador, por ejemplo `LFG_cgi2121.hs`, el manejador de la página "editar entrada léxica" puede devolver la página principal se apretó el botón "volver a la página principal", este es uno de los motivos de la división entre páginas y manejadores).

4.4.3 Ciclo de trabajo

Primeramente se accede al sitio:

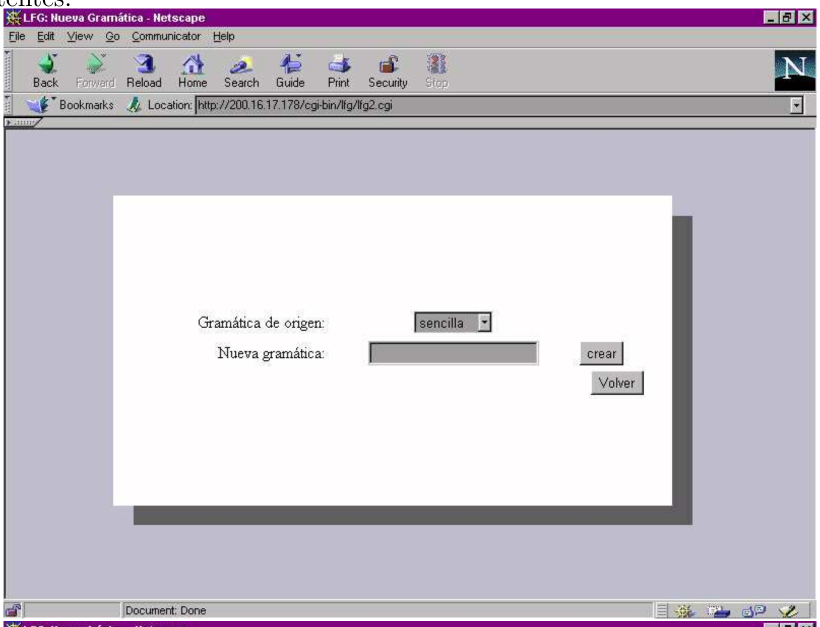


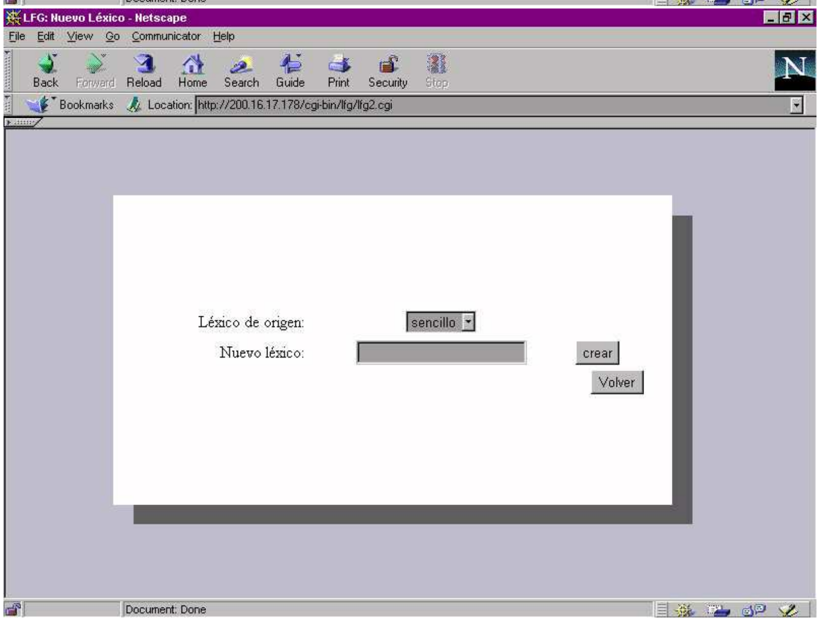
Luego tenemos el menú principal:



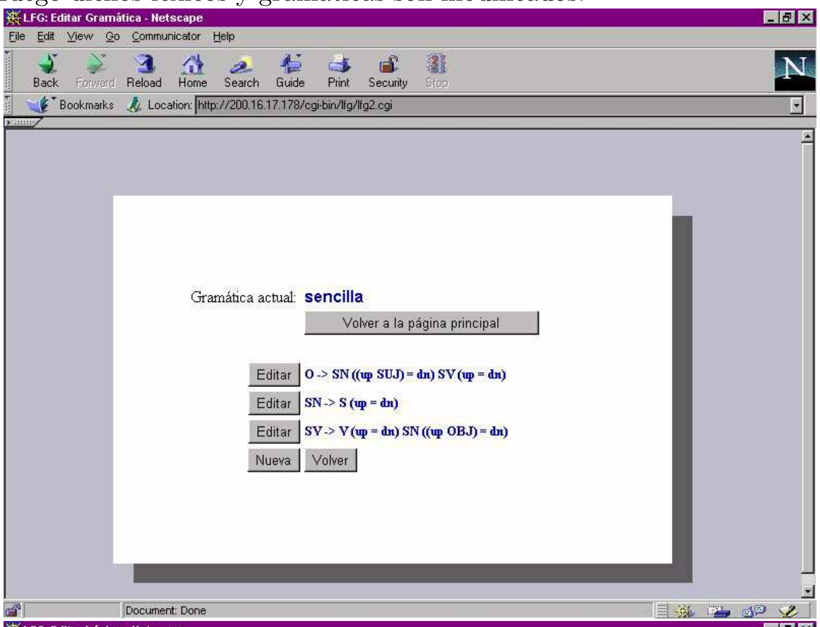
en el cual podemos cambiar la gramática y el léxico de trabajo.

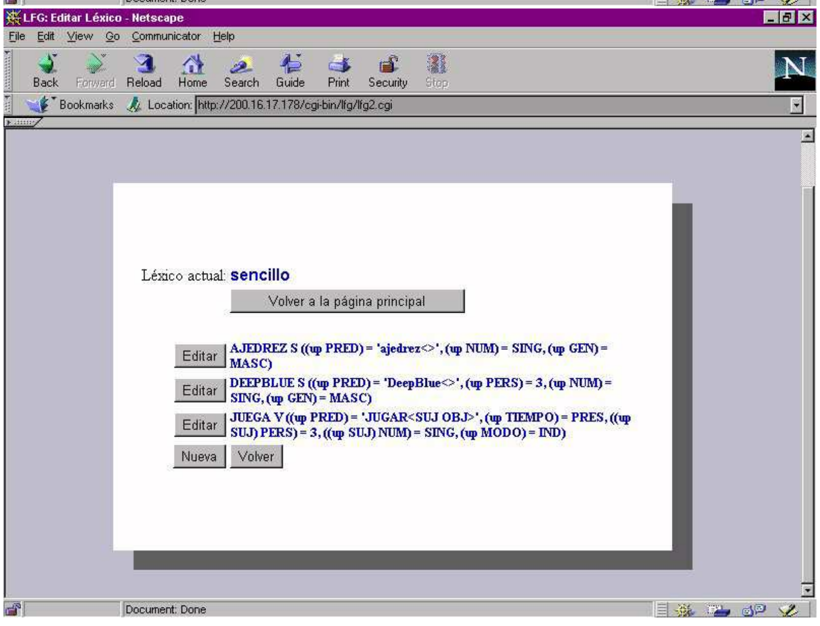
Generalmente se crean nuevas gramáticas y léxicos, copiando desde los ya existentes:



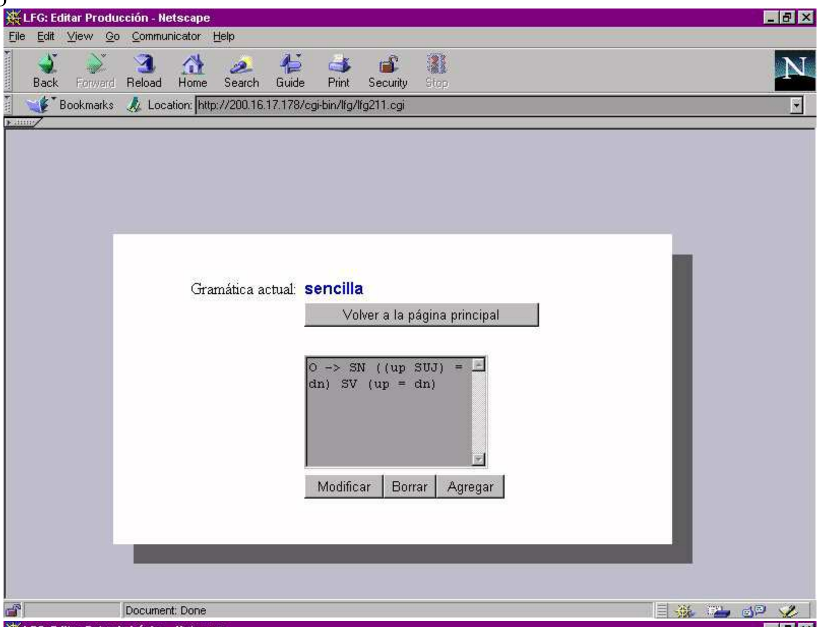


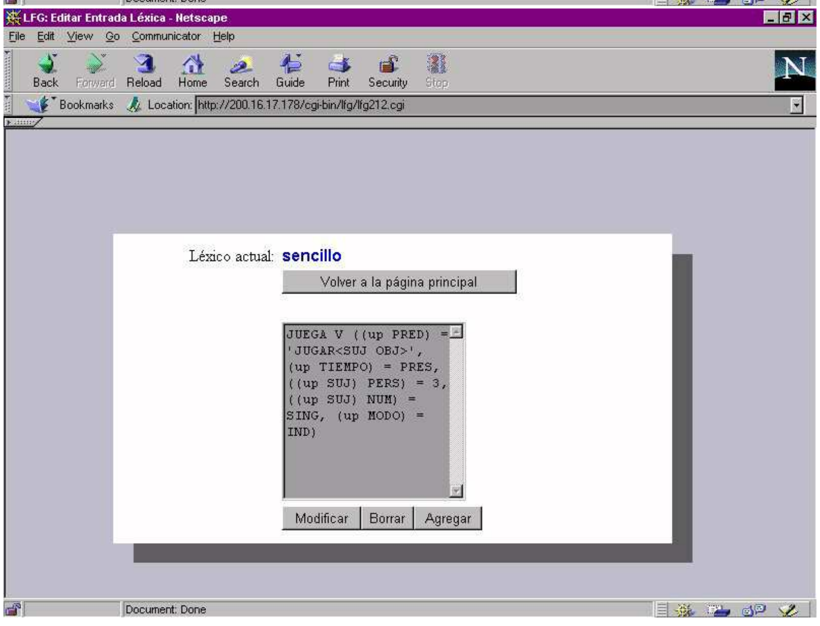
Luego dichos léxicos y gramáticas son modificados:



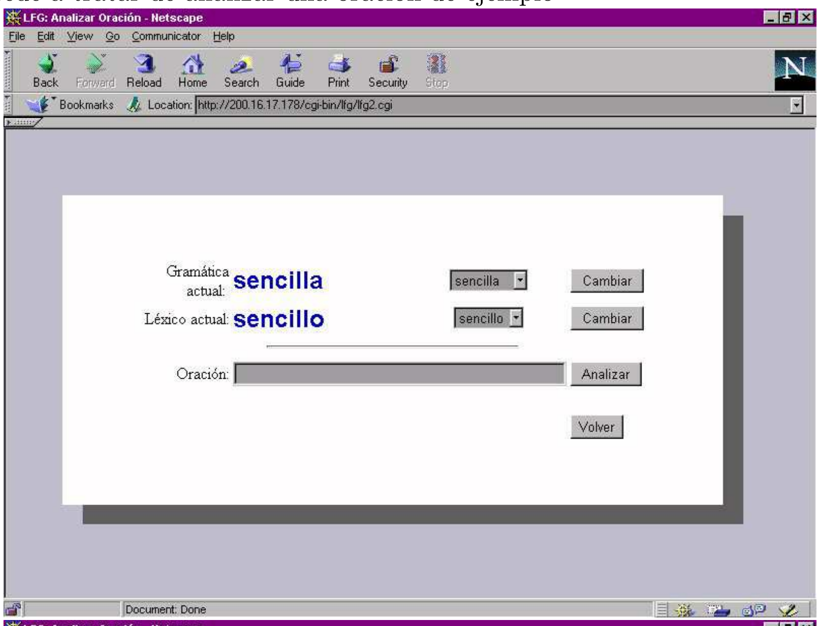


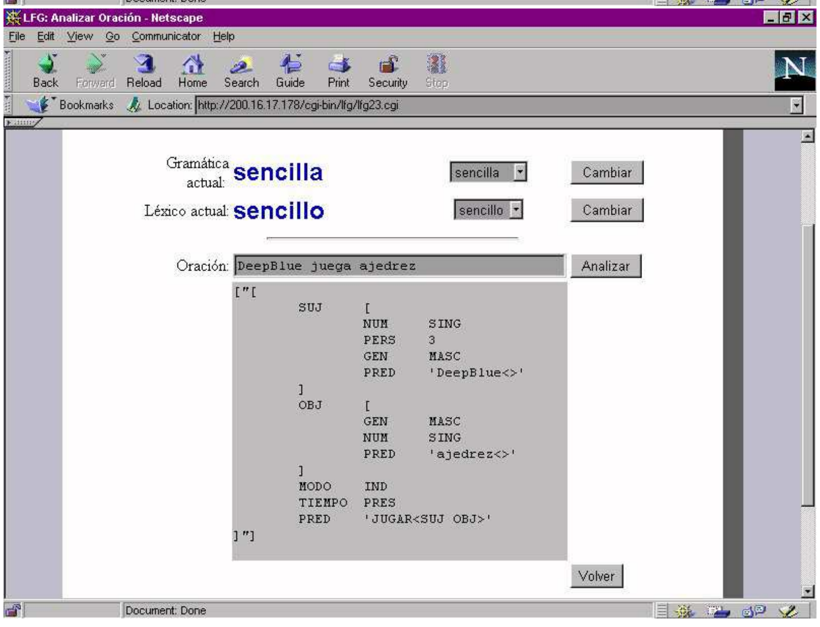
editando, agregando o borrando producciones y entradas léxicas, según sea necesario





Una vez la gramática y el léxico reflejan los intereses del experimento se procede a tratar de analizar una oración de ejemplo





Es interesante el hecho de que en “analizar oración” podemos cambiar de léxico y de gramática y mantener la misma oración, de forma tal que resulte muy cómodo experimentar con gramáticas y léxicos con pequeñas diferencias.

Capítulo 5

El castellano

5.1 Resultados

Para acotar la longitud del proyecto nos fué sugerido utilizar un párrafo extraído de algún texto real en castellano (dicha sugerencia provino desde el área de lingüística, por lo que entendimos que el ejercicio de analizar tan sólo un pequeño trozo de texto tenía un cierto interés). Se utilizó una nota periodística aparecida en la revista **Ecos de la ciudad** número 76, que habla acerca del área natural protegida de la bahía de San Antonio. En dicha nota puede verse en un recuadro titulado "peligros latentes" la siguiente oración:

(5.1)
"Más allá de los datos que marcan cambios significativos en el tamaño poblacional de numerosas especies, el problema está en la progresiva intromisión de los seres humanos en sus habitats."

Utilizando este texto como meta se realizó la siguiente gramática:

(5.2)

$MODIF$	$\rightarrow$	$ADV\ (\uparrow = \downarrow)$
$MODIF$	$\rightarrow$	$ADV\ (\uparrow = \downarrow)\ SP\ ((\uparrow DET) = \downarrow)$
O	$\rightarrow$	$MODIF\ ((\uparrow MODIF) = \downarrow)\ COMA\ (\uparrow = \downarrow)$
		$SN\ ((\uparrow SUJ) = \downarrow)\ SV\ (\uparrow = \downarrow)$
O	$\rightarrow$	$SN\ ((\uparrow SUJ) = \downarrow)\ SV\ (\uparrow = \downarrow)$
OS	$\rightarrow$	$PR\ ((\uparrow \text{SUJ}) = \downarrow)\ \text{SV}\ (\uparrow = \downarrow)$
SN	$\rightarrow$	$ART\ (\uparrow = \downarrow)\ A\ (\uparrow = \downarrow)\ S\ (\uparrow = \downarrow)$
SN	$\rightarrow$	$ART\ (\uparrow = \downarrow)\ A\ (\uparrow = \downarrow)\ S\ (\uparrow = \downarrow)$
		$SP\ ((\uparrow DET) = \downarrow)$
SN	$\rightarrow$	$ART\ (\uparrow = \downarrow)\ A\ (\uparrow = \downarrow)\ S\ (\uparrow = \downarrow)$
		$SP\ ((\uparrow DET) = \downarrow)\ SP\ ((\uparrow DET2) = \downarrow)$
SN	$\rightarrow$	$ART\ (\uparrow = \downarrow)\ S\ (\uparrow = \downarrow)$
SN	$\rightarrow$	$ART\ (\uparrow = \downarrow)\ S\ (\uparrow = \downarrow)\ A\ (\uparrow = \downarrow)$
SN	$\rightarrow$	$ART\ (\uparrow = \downarrow)\ \text{S}\ (\uparrow = \downarrow)\ A\ (\uparrow = \downarrow)$
		$SP\ ((\uparrow DET) = \downarrow)$
SN	$\rightarrow$	$ART\ (\uparrow = \downarrow)\ S\ (\uparrow = \downarrow)\ OS\ ((\uparrow DET) = \downarrow)$
SN	$\rightarrow$	$S\ (\uparrow = \downarrow)\ A\ (\uparrow = \downarrow)$
SP	$\rightarrow$	$P\ (\uparrow = \downarrow)\ SN\ ((\uparrow OBJ) = \downarrow)$
SV	$\rightarrow$	$V\ (\uparrow = \downarrow)$
SV	$\rightarrow$	$V\ (\uparrow = \downarrow)\ SN\ ((\uparrow OBJ) = \downarrow)$
SV	$\rightarrow$	$V\ (\uparrow = \downarrow)\ SN\ ((\uparrow OBJ) = \downarrow)$
		$SP\ ((\uparrow OBL) = \downarrow)$
SV	$\rightarrow$	$V\ (\uparrow = \downarrow)\ SP\ ((\uparrow OBL) = \downarrow)$
V	$\rightarrow$	$ADV\ ((\uparrow MODIF) = \downarrow)\ V\ (\uparrow = \downarrow)$

Y el siguiente léxico:

(5.3)

humanos	A	$((\uparrow GEN) = MASC,$
	$(\uparrow NUM) = PL,$	
	$(\uparrow HUMANO) = SI)$	
numerosas	A	$((\uparrow GEN) = FEM,$
	$(\uparrow NUM) = PL,$	
	$(\uparrow NUMEROSO) = SI)$	
poblacional	A	$((\uparrow NUM) = SING,$
	$(\uparrow DEPOBLACION) = SI)$	
progresiva	A	$((\uparrow GEN) = FEM,$
	$(\uparrow NUM) = SING,$	
	$(\uparrow PROGRESIVO) = SI)$	
significativos	A	$((\uparrow GEN) = MASC,$
	$(\uparrow NUM) = PL,$	
	$(\uparrow IMPORTANTE) = SI)$	

(5.4)

más_allá ADV $((\uparrow PRED) = \text{'MASALLA'})$

ART $((\uparrow PERS) = 3,$
$(\uparrow NUM) = SING,$
$(\uparrow GEN) = MASC,$
$(\uparrow DETERM) = +)$

ART $((\uparrow PERS) = 3,$
$(\uparrow GEN) = FEM,$
$(\uparrow NUM) = SING,$
$(\uparrow DETERM) = +)$

las ART $((\uparrow PERS) = 3,$
$(\uparrow NUM) = PL,$
$(\uparrow GEN) = FEM,$
$(\uparrow DETERM) = +)$

los ART $((\uparrow PERS) = 3,$
$(\uparrow NUM) = PL,$
$(\uparrow GEN) = MASC,$
$(\uparrow DETERM) = +)$

sus ART $((\uparrow PERS) = 3,$
$(\uparrow NUM) = PL)$

$COMA$ ()

$((\uparrow PRED) = \text{'PERTENENCIA}<(\uparrow OBJ)>\text{'})$

$((\uparrow PRED) = \text{'LUGAR}<(\uparrow OBJ)>\text{'})$

que PR $((\uparrow PRED) = \text{'PROREL'})$

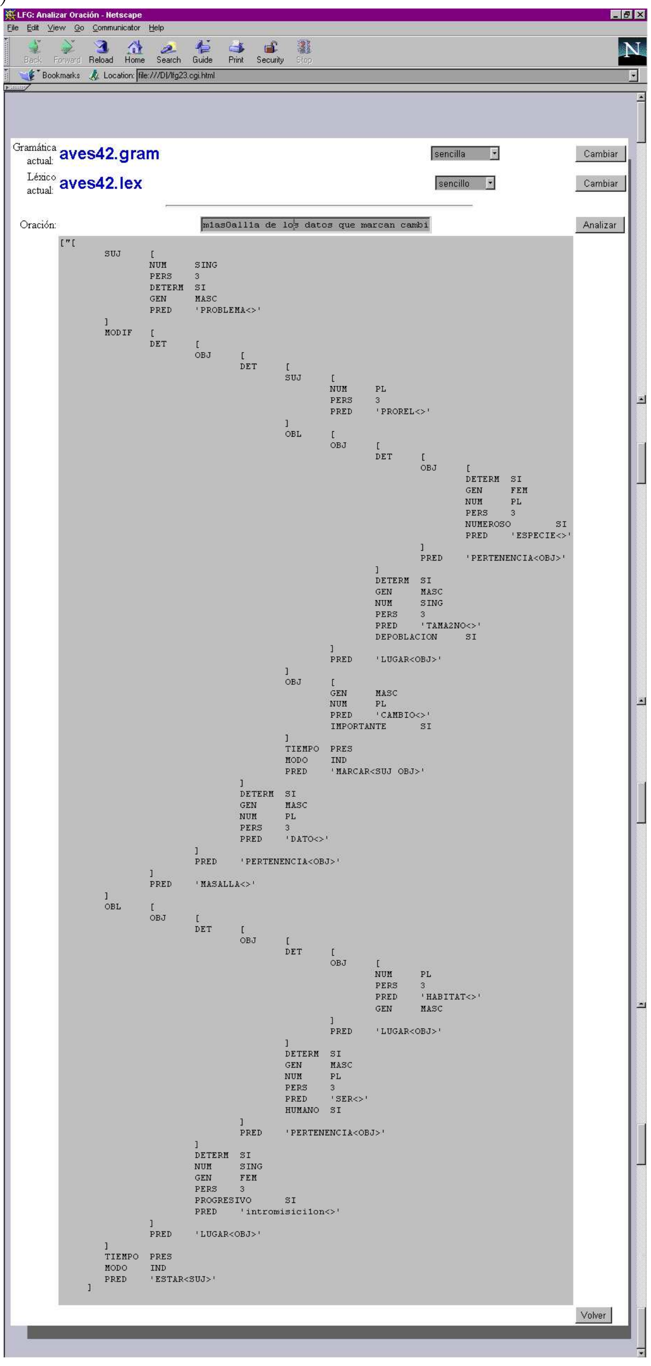
cambios S $((\uparrow PRED) = \text{'CAMBIO'},$
$(\uparrow PERS) = 3,$
$(\uparrow NUM) = PL,$
$(\uparrow GEN) = MASC)$

datos S $((\uparrow PRED) = \text{'DATO'},$
$(\uparrow PERS) = 3,$
$(\uparrow NUM) = PL,$
$(\uparrow GEN) = MASC)$

(5.5)

especies	S $(\uparrow PERS) = 3,$ $(\uparrow NUM) = PL,$ $(\uparrow GEN) = FEM)$	$((\uparrow PRED) =$ 'ESPECIE',
habitats	S $(\uparrow PERS) = 3,$ $(\uparrow NUM) = PL,$ $(\uparrow GEN) = MASC)$	$((\uparrow PRED) =$ 'HABITAT',
intromisión	 $(\uparrow PERS) = 3,$ $(\uparrow NUM) = SING,$ $(\uparrow GEN) = FEM)$	$((\uparrow PRED) =$ 'INTROMISIÓN',
problema	S $(\uparrow PERS) = 3,$ $(\uparrow NUM) = SING,$ $(\uparrow GEN) = MASC)$	$((\uparrow \text{PRED}) =$ 'PROBLEMA',
seres	$((\uparrow PRED) =$ 'SER', $(\uparrow PERS) = 3,$ $(\uparrow NUM) = PL,$ $(\uparrow GEN) = MASC)$	
tamaño	S $(\uparrow PERS) = 3,$ $(\uparrow NUM) = SING,$ $(\uparrow GEN) = MASC)$	$((\uparrow PRED) =$ 'TAMAÑO',
está V	$((\uparrow PRED) = \text{'ESTAR}<(\uparrow SUJ)>\text{'},$ $((\uparrow SUJ)\ PERS) = 3,$ $(\uparrow MODO) = IND,$ $(\uparrow TIEMPO) = PRES,$ $((\uparrow SUJ)\ NUM) = SING)$	
marcan V	$((\uparrow PRED) = \text{'MARCAR}<(\uparrow SUJ)\ (\uparrow OBJ)>\text{'},$ $((\uparrow SUJ)\ PERS) = 3,$ $(\uparrow MODO) = IND,$ $(\uparrow TIEMPO) = PRES,$ $((\uparrow SUJ)\ NUM) = PL)$	

A continuación puede observarse la salida del programa con el análisis sintáctico de (5.1)



Esta pareja gramática–léxico tiene las siguientes particularidades:

1. Concordancia sujeto–verbo en persona y número.
2. Concordancia adjetivo–sustantivo en género y número.
3. Concordancia artículo–sustantivo en género y número.
4. Verificación de subcategorización para verbos transitivos e intransitivos.

Concordancia sujeto–verbo

Siguiendo los lineamientos generales de ejemplos en idioma inglés, la concordancia sujeto–verbo en número y persona puede resolverse a nivel léxico, agregando en la entrada de cada verbo dos ecuaciones como en el siguiente ejemplo:

(5.6)

V $((\uparrow PRED) =$ 'VER<$(\uparrow SUJ)$ $(\uparrow OBJ)$>',
$((\uparrow SUJ)\ PERS) = 3$,
$((\uparrow SUJ)\ NUM) = SING)$

Estas ecuaciones determinarán, de este modo, que si el sujeto no está por ejemplo en singular, entonces queden ecuaciones inconsistentes para $SUJ\ NUM$, prediciéndose correctamente a la oración como no perteneciente al castellano. Ahora bien, para que el chequeo de concordancia sea posible, es importante que en el SN estén presentes el género y persona correspondientes.

Concordancia adjetivo–sustantivo

En castellano los adjetivos deben concordar con el sustantivo o SN al que modifican en género y número. En los textos consultados no encontramos ninguna solución a este problema pues éstos se centraban principalmente en el idioma inglés, el cual no presenta tal concordancia.

Se estudiaron distintas alternativas. En primer lugar, se pensó en utilizar la misma solución que la empleada en el caso de los verbos, esto es, resolverlo a nivel léxico. Pero para utilizar **exactamente** la misma aproximación era necesario que el sustantivo o el SN estuviera **inserto** dentro de la estructura-f del adjetivo como una subestructura. Es decir:

(5.7)

progresiva A $((\uparrow PRED) =$ 'PROGRESIVA<$(\uparrow OBJ)$>',
$((\uparrow OBJ)\ GEN) = FEM$,
$((\uparrow OBJ)\ NUM) = SING)$

intromisión S $((\uparrow PRED) =$ 'INTROMISIÓN',
$(\uparrow PERS) = 3$,
$(\uparrow GEN) = FEM$,
$(\uparrow NUM) = SING)$

con gramática:

(5.8)

$SN \quad \rightarrow \quad A\ (\uparrow = \downarrow)\ S\ ((\uparrow OBJ) = \downarrow)$

Asi el SN *progresiva intromisión* generaría la siguiente estructura-f:

$$(5.9) \quad \begin{bmatrix} OBJ & \begin{bmatrix} PRED & \text{`INTROMISIÓN'} \\ NUM & SING \\ GEN & FEM \\ PERS & 3 \end{bmatrix} \\ PRED & \text{`PROGRESIVA} < (\uparrow\ OBJ) >\text{'} \end{bmatrix}$$

la cual contiene toda la información sintáctica del SN pero presenta la siguiente falencia importante: la información del S ha quedado "escondida" dentro de una subestructura. De este modo el método anteriormente mencionado para obtener concordancia sujeto–verbo ya no será aplicable. Además, de tener varios adjetivos, la información del S estará aun más profunda en la estructura. Creemos que esto no caracteriza la importancia del núcleo en un SN por lo que no nos parece un buen modelo.

Otra alternativa era utilizar reglas que no fuesen del tipo visto, por ejemplo con construcciones del tipo:

(5.10)

$SN \quad \rightarrow \quad A\ ((\uparrow DET) = \downarrow, (\uparrow GEN) = (\downarrow GEN))\ S\ (\uparrow = \downarrow)$

Pero, por un lado parece ser que el tipo de ecuaciones al que nos hemos restringido es suficiente para un número grande de aplicaciones. Por otra parte, habría que estudiar que cambios en `resolverF` acarrearía estad decisión. Por ello se lo dejó para trabajos futuros.

La alternativa que se utilizó finalmente es una solución pero no totalmente: consiste en suponer a nuestro sistema como dentro de un determinado 'mundo' en el cual los adjetivos aportan **características** a los sustantivos, características que pueden expresarse por medio de funciones gramaticales. Esto es, para el ejemplo anterior:

(5.11)

progresiva $\quad A \quad ((\uparrow GRADUAL) = +,$
$((\uparrow OBJ)\ GEN) = FEM,$
$((\uparrow OBJ)\ NUM) = SING)$

con gramática:

(5.12)

$SN \quad \rightarrow \quad A\ (\uparrow = \downarrow)\ S\ (\uparrow = \downarrow)$

¿Cuál es el problema de está aproximación? Contradice en algún sentido lineamientos generales de LFG (y, tal vez, lingüísticos) pues estamos poniendo semántica en las funciones gramaticales (y no en los $PRED$, donde pertenecen). Además del hecho que las funciones gramaticales se suponen en número acotado y en algún sentido universales.

Sin embargo este enfoque puede ser útil pues dependiendo del lenguaje que estemos analizando es posible que la cantidad de adjetivos de interés sea muy pequeña y su función modificadora quede bien determianda de este modo. El lenguaje literario suele estar poblado de adjetivos y formas adornadas pero otros lenguajes, como sería el caso de una aplicación de mandos por voz, seguramente involucrará un número mucho menor de adjetivos en su léxico.

Subcategorización de objetos directos

Con la adición de la función `validarF` el sistema es capaz de rechazar oraciones en las cuales la forma semántica del verbo requiera un OBJ y éste no esté presente, por un lado, u oraciones con un OBJ presente y no requerido, por el otro.

Con este método prodría también exigirse la ausencia de un sujeto en verbos impersonales, como por ejemplo *llover*.

También es posible detectar qué variante del verbo está en uso, tomemos el caso de *comer* con dos acepciones, una transitiva y la otra intransitiva. Cuando lleva un OBJ se lo considera equivalente a la acción de llevar algo a la boca, introducirlo, posiblemente masticarlo y luego tragarlo. Si no lleva OBJ es sinónimo de *alimentarse*. Esto podría ser representado en un léxico LFG del siguiente modo:

(5.13)

$$\begin{array}{ll}
\text{come } V & ((\uparrow PRED) = \text{'COMER1}<(\uparrow SUJ)\ (\uparrow OBJ)>\text{'}, \\
 & ((\uparrow SUJ)\ PERS) = 3, \\
 & ((\uparrow SUJ)\ NUM) = SING) \\
 & \\
\text{come } V & ((\uparrow PRED) = \text{'COMER2}<(\uparrow SUJ)>\text{'}, \\
 & ((\uparrow SUJ)\ PERS) = 3, \\
 & ((\uparrow SUJ)\ NUM) = SING)
\end{array}$$

Tendríamos asi que para la oración *Juan come* el sistema determinará que la forma semántica correspondiente es 'COMER2' que no subcategoriza a un objeto directo.

5.2 Conclusiones

Las conclusiones son bastante buenas. Al empezar el estudio de un campo tan aparentemente alejado de las Cs. de la Computación como es la lingüística el trabajo parecía infinito. A pesar de esto, los textos de dicha área nos muestran como se avanza lentamente hacia el conocimiento más o menos completo del lenguaje. Este proceso es netamente experimental: se construyen modelos y se los destruye constantemente, siempre tratando de explicar más particularidades y no perder los logros ya alcanzados. Evidentemente en este proceso muchas veces el modelo se muestra insuficiente, en cuyo caso se reforma o se abandona por completo, dependiendo de la magnitud de los embates que haya recibido.

Las Cs. de la Computación por un lado acompaña este proceso proveyendo herramientas que permitan trabajar de manera más fácil con los modelos, y, por otro, genera sus propias herramientas tal vez sobre casos de menor valor lingüístico pero que permitan, por ejemplo, una mejora en las interfaces con el usuario, resumen automático, traducción asistida, etcétera.

En ese sentido, creemos que la herramienta desarrollada cumple con ambos objetivos: Desde el punto de vista de la lingüística , se espera sea útil para realizar tareas de investigación en LFG en el sentido que permite probar y confrontar distintas hipótesis sintáctias. Y para las Cs. de la Computación un analizador sintáctico de este tipo puede ser el cimiento de un sin número de aplicaciones en lingüística computacional, por ejemplo, basta cambiar el tipo de las formas semánticas para permitir que los verbos incluyan funciones (esto es posible en lenguajes funcionales, claro está) y los sustantivos y otras clases de palabras incluyan tipos de datos estructurados; para obtener estructuras-f "ejecutables" y, por ejemplo, mandos por oraciones. Uniendo esto a un sistema de reconomiento del habla podemos tener mandos por voz. Sino, uniéndolo a un motor de juegos de aventuras podemos tener un juego de aventuras conversacionales (al estilo del viejo *adventure*). Sino podemos trabajar con la estructura-f 'podándola', esto es, quitando subestructuras-f muy profundas, con la esperanza que estén marcando conceptos secundarios, uniendo esto a un generador de texto obtendríamos un comienzo de un resumidor automático (un detalle, de la estructura-f el generador extraería una oración en un formato más sencillo, simplificando la lectura y facilitando la comprensión).

Otro punto importante respecto del trabajo realizado está centrado en la utilización de un lenguaje funcional puro de evaluación perezosa. La elección del paradigma funcional representado por Haskell frente a un mucho más tradicional paradigma lógico, programado generalmente en Prolog demostró ser el camino correcto. No sólo resulta más fácil razonar sobre los programas en funcional sino que el código obtenido es razonablemente eficiente sin perder transparencia y la posibilidad de demostrar los algoritmos.

Los problemas usuales de los lenguajes funcionales puros, tales como ineficiencia en la ejecución y dificultades para realizar entrada/salida no fueron un escollo insalvable en el presente trabajo. Ciertamente la tecnología en materia de compiladores de lenguajes funcionales ya está lo suficientemente madura como para comenzar a competir con sus contrapartes imperativas.

El sistema presentado hace un uso extensivo de las ventajas propias de este paradigma: durante su desarrollo muchas decisiones de diseño fueron cambiadas, inclusive la adición de una interface vía WWW no fué tomada desde el principio, lo cual no significó mayores inconvenientes. El hecho de poder razonar ecuacionalmente sobre los programas permite rápidamente comprender, analizar y depurar código escrito a veces hace intervalos de tiempo importantes. Sin contar con que la ausencia de efectos secundarios permite aislar con precisión los errores.

Este desarrollo se vió muy beneficiado también por el sistema de módulos del lenguaje Haskell. Las características avanzadas presente en dicho sistema modular tales como renombre, ocultamiento y "calificación" hicieron posible la posterior inclusión de librerías escritas por terceros que incluían en general nombres que estan en oposición con nombres del sistema.

Finalmente, algunas palabras sobre el formalismo lingüístico empleado y, en particular sobre el castellano. El formalismo de las gramáticas léxico funcionales ha resultado ciertamente muy poderoso. Su implementación presenta algunos inconvenientes y puntos oscuros, sobre todo en torno a la unificación pero una vez "en producción" es muy fácil razonar en él y permite analizar oraciones que a priori parecen intratables.

Sin embargo, se implementó la versión original de LFG la cual posteriormente fue mejorada. Dentro de los trabajos futuros está proyectado expandir el sistema con las funciones avanzadas de la teoría.

Un tema aparte es el del Castellano. Siguiendo a [13], ciertamente el idioma español es de las lenguas romances la que más conserva la falta de *linearidad* latina, otras lenguas romances al perder las desinencias son más exigentes a la hora de determinar el orden de las palabras en la oración.

Las experiencias realizadas parecen indicar que una caracterización a nivel de gramática libre de contexto de todas las posibles variaciones en el orden de los sintagmas que encontramos en castellano llevaría a un resultado final que en el caso promedio devolvería demasiados *parsings* posibles (esto es, la gramática sería demasiado ambigua y sin la obligada inclusión de un **buen** módulo semántico el resultado del analizador carecería de utilidad). Ahora bien, para numerosas aplicaciones con lenguaje restringido podemos suponer que las personas utilizarán un lenguaje más "lineal" pues también estarán interesadas en poder "comunicarse" con la máquina.

5.3 Trabajos futuros

El presente trabajo está abierto a muchas extensiones:

En el plano de la lingüística, como dijimos antes, incorporar los últimos avances en LFG pondría a la herramienta en una posición más actualizada, además de mejorar su campo de acción. Algunos de los cambios en ese sentido son muy sencillos de realizar como la inclusión de ecuaciones restrictivas. Otros, por el contrario, es muy posible que introduzcan reestructuraciones en algunos algortimos. De cualquier modo confiamos en que las estructuras de datos y las abstracciones diseñadas son suficientemente buenas para capturar las nociones involucradas en la teoría.

En el plano de la programación funcional habría que avanzar en cerrar el código a modo de *dinamic link library* o más bien en el sentido de una "caja negra" analizadora, permitiendo la utilización del motor desde lenguajes imperativos tradicionales (e.g., C).

Un primer paso en ese sentido sería cerrar el sistema no ya con la interface WWW sino como un *daemon* analizador (suponiendo que la gramática y el léxico ya están "en producción"). De este modo, mediante el uso de *sockets* y las extensiones `posix` del haskell se tendría en algún puerto dado y con un protocolo sencillo un servicio de análisis sintáctico. Esto nos libraría de problemas de plataforma (pues actualmente el sistema corre bajo Linux).

Otra posibilidad serí no librarse de los problemas de plataforma sino, por el contrario, enfrentarlos y migrar el código a entorno Win32. De nuevo aquí el hecho de haber utilizado haskell nos resta inconvenientes. Es de esperar que la virtualización que nos ofrece el compilador signifique un esfuerzo para realizar la migración no mayor a instalar la versión Win32 del GHC (esto es, esperamos que el código no necesite ser modificado para tal fin). De ser necesario también puede hacerse un *port* a MacOs.

Un detalle que puede seguirse avanzando respecto del código funcional es en la utilización de las excepciones de la mónada `IO`, las cuales actualmente apenas se usan. Esto permitiría mejoran sustancialmente la robustez del sistema.

Un tema pendiente que no se pasó por alto pero que no se pudo concretar es el de la demostración de los algoritmos. Funcional hace muchísimo más sencilla esta tarea pero de cualquier modo las demostraciones de algoritmos no triviales son siempre engorrosas. El algoritmo más interesante de demostrar es el de la unificación, `resolverF`. Ya se dieron algunos pasos en demostrar su correctitud (esto es, toda la solución devuelta por él es solución del sistema). Aparte estamos interesado en probar su completitud (toda solución es alcanzada por él) y minimalidad (las soluciones generadas son las más pequeñas).

Estas demostraciones no son sencillas pero apoyándonos en los resultados usuales de los trabajos en unificación [19], confiamos llevarlas a buen puerto en el futuro próximo.

Respecto de la lingüística computacional, quedan grandes desafios: En primer lugar está el hecho de programar un mejor *lexer*, con reglas, desinencias, conjugaciones verbales, etcétera. Otro punto es permitir gramáticas recursivas a izquierda como entrada al algoritmo. Para ello se aplican transformaciones de gramáticas. La pregunta aquí es *¿Qué pasa, cuando trasnformo una gramática, con los esquemas funcionales?*. La idea es buscar la respuesta a tales cuestiones en las gramáticas con atributos que aparecen normalmente en los compiladores.

Relacionado con el tema de la transformación de gramáticas y compiladores, otro punto en el cual se puede seguir el trabajo es en la realización del analizador sintáctico ahora con un enfoque **compilado** esto es, hacer un programa que reciba como parámetros la gramática y genere código funcional que reconozca dicha gramática, siguiendo los lineamientos de [24]. Esto podría juntarse a la idea de generar el "daemon de análisis sintáctico" anteriormente descriptas.

Finalmente está el hecho de buscar aplicaciones para el analizador, desarrollando los módulos que fuesen necesarios.

Bibliografía

[1] Aho, A., *Indexed grammars - an extension of context-free grammars* (Journal of the ACM, 1968)

[2] Aho, A., Sethi, R., Ullman, J., *Compiladores (Principios, técnicas y herramientas)*

(Addison-Wesley iberoamericana, 1986)

[3] Akmajian, A., Demers, R., Farmer, A., Harnish, R., *Linguistics (An introduction to Language and Communication)*

(MIT Press, Third edition, 1992)

[4] Allen, J., *Natural Language Understanding*

(Benjamin/Cummig, 1987)

[5] Amores, J., Quesada, J., *Lekta: A tool for the development of efficient LFG-based Machine Translation systems*

(1995)

[6] Amores, J., Quesada, J., *Lekta User Manual (Version 1.0)*

(1995)

[7] Bianchi, C., *GETARUN*

`http://aida.eng.unipr.it/cgi-bin/bianchi/cgi-bin/getarun-1.1/pl3w_interprete?fai_frames`

[8] Bird, R., Wadler, P., *Introduction to Functional Programming*

(Prentice Hall, 1988)

[9] Chomsky, N., *Syntactic Structures*

(The Hague: Mouton, 1957)

[10] Chomsky, N., *Reflections on language*

(New York: Pantheon Books, 1975)

[11] Davis, M., Weyuker, E., *Computability, Complexity and Languages*

(Academic Press, 1983)

[12] Gadzar, G., Pullum, G., *Computationally Relevant Properties of Natural Languages and Their Grammars*

(Center for the Study of Languages and Information, 1985)

[13] Gili Gaya, S. *Curso superior de Sintaxis Española* VOX, Decimoquinta impresión, 1994

[14] Hopcroft, J., Ullman, J., *Introduction to Automata Theory, Languages and Computation*

(Addison-Wesley, 1979)

[15] Hudak, P., Peyton Jones, S., Wadler, P., et al., *Haskell, Report on the Programming Language*

(1992)

[16] Hutton, G., Meijer, E., *Monadic Parsing in Haskell*

(Journal of Functional Programming, 1993)

[17] Hutton, G., Meijer, E., *Monadic Parser Combinators*

(1996)

[18] Johnson, S., *YACC –yet another compiler compiler* (Bell Laboratories, 1975)

[19] Jounannaud, J., Kirchner, C., *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification*, Festschrift for Alan Robinson, MIT Press

[20] Kaplan, R., *On Process Models for Sentence Comprehension*

(Explorations in congnition, San Francisco: W.H. Freeman, 1975)

[21] Kaplan, R., Bresnan, J., *Lexical-Functional Grammar: A Formal System for Grammatical Representation*

(Cambridge, 1982)

[22] Kaplan, R., *The Formal Architecture of Lexical-Functional Grammar*

(Formal issues in Lexical-Functional Grammar, 1994)

[23] Lambek, J., Scott, P., *Introduction to Higher Order Categorical Logic* (Cambridge University Press, 1986)

[24] Leermakers, R., *The Functional Treatment of Parsing*

(1994)

[25] Mac Lane, S., *Categories for the Working Mathematician* (Springer-Verlag, 1971)

[26] Meijer, E., *CGI Programming* `http://www.cse.ogi.edu/~erik/Personal/cgi.htm`

[27] Miller, G., Gildea, P., *How childrean learn words* (Scientific American, 1987)

[28] Moggi, E., *Computational lambda-calculus and monads* (IEEE, 1989)

[29] Moggi, E., *An abstrac view of programming languages* (University of Edinburgh, 1989)

[30] Neidle, C., *Lexical Functional Grammar*

(1991)

[31] Pereira, F., *A new characterization of attachment preferences*
(Cambridge University Press, 1985)

[32] Peyton Jones, S., *The Implementation of Functional Programming Languages*
(Prentice Hall, 1987)

[33] Sadler, L., *New Developments in Lexical Functional Grammar*
(1996)

[34] Sells, P., *Teorías sintácticas actuales (GB, GPSG, LFG)*
(Editorial Teide, 1985)

[35] Shieber, S., *Sentence disambiguation by a shift-reduce parsing technique*
(Proc. of the 21st Annual Meeting of the Association for Computational Linguistics, 1983)

[36] Shieber, S., *Evidence against the non-context-freeness of natural language*
(Linguistics and Philosophy, 1985)

[37] Spivey, M., *A functional theory of exceptions* (Sicence of Computer Programming, 1990)

[38] Tomita, M., *LR parsers for natural languages*
(Proc. of the Coling84, 1984)

[39] Wadler, P., *Comprehending Monads*
(Proc. ACM conference on Lisp and functional programming, 1991)

[40] Wadler, P., *The essence of functional programming* (Proc. principles of programming languages, 1992)

[41] Wadler, P., *Mondas for functional programming* (Springer Verlag, 1992)

[42] Wescoat, M. *Practical Instructions for Working with the Formalism of Lexical Functional Grammar*
(1985)

[43] Woods, W., *Transition Network Grammars for Natural Language Analysis* (Communications of the ACM, 13(10), 1970)