Syntax and Parsing

COMS W4115

Prof. Stephen A. Edwards
Spring 2002
Columbia University
Department of Computer Science
Last Time

Administrivia

Class Project

Types of Programming Languages:

Imperative, Object-Oriented, Functional, Logic, Dataflow
This Time

Interpreters and Compilers
Structure of a Compiler
Lexical Analysis
Syntax
Parsing
The Compilation Process
Interpreters

Source Program

\[\text{Input} \rightarrow \text{Interpreter} \rightarrow \text{Output}\]
Compilers

Source Program

↓

Compiler

↓

Input  →  Executable Program  →  Output
Structure of a Compiler

Program Text

Lexer

Token Stream

Parser

Static semantics (type checking)

Abstract Syntax Tree

Translation to intermediate form

Annotated AST

Code generation

Three-address code

Assembly Code
Compiling a Simple Program

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```
What the Compiler Sees

```c
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

Text file is a sequence of characters
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}

A stream of tokens. Whitespace, comments removed.
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}

Abstract syntax tree built from parsing rules.
After Semantic Analysis

Symbol Table:

```
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
</tr>
</tbody>
</table>
```

Types checked; references to symbols resolved
After Translation into 3-Address Code

L0: sne $1, a, b
seq $0, $1, 0
btrue $0, L1 % while (a != b)
sl $3, b, a
seq $2, $3, 0
btrue $2, L4 % if (a < b)
sub a, a, b % a -= b
jmp L5

L4: sub b, b, a % b -= a
L5: jmp L0
L1: ret a

Idealized assembly language w/ infinite registers

int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
After Translation to 80386 Assembly

gcd: pushl %ebp % Save frame pointer
    movl %esp,%ebp
    movl 8(%ebp),%eax % Load a from stack
    movl 12(%ebp),%edx % Load b from stack
.L8: cmpl %edx,%eax % while (a != b)
                je .L3 % if (a < b)
                jle .L5 % a -= b
    subl %edx,%eax
    jmp .L8
.L5: subl %eax,%edx % b -= a
    jmp .L8
.L3: leave % Restore SP, BP
    ret
Lexical Analysis (Scanning)
Lexical Analysis (Scanning)

Goal is to translate a stream of characters

\[ \text{int } \text{sp gcd } \text{int } \text{sp a, int sp b} \]

into a stream of tokens

<table>
<thead>
<tr>
<th>ID</th>
<th>ID</th>
<th>LPAREN</th>
<th>ID</th>
<th>ID</th>
<th>COMMA</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>gcd</td>
<td></td>
<td>int</td>
<td>a</td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each token consists of a token type and its text.

Whitespace and comments are discarded.
Lexical Analysis

Goal: simplify the job of the parser.

Scanners are usually much faster than parsers.

Discard as many irrelevant details as possible (e.g., whitespace, comments).

Parser does not care that the identifier is “supercalifragilisticexpialidocious.”

Parser rules are only concerned with token types.
The ANTLR Compiler Generator

Language and compiler for writing compilers

Running ANTLR on an ANTLR file produces Java source files that can be compiled and run.

ANTLR can generate

- Scanners (lexical analyzers)
- Parsers
- Tree walkers

We will use all of these facilities in this class
An ANTLR File for a Simple Scanner

class CalcLexer extends Lexer;

LPAREN : '(' ; // Rules for punctuation
RPAREN : ')';
STAR : '*' ;
PLUS : '+' ;
SEMI : ';' ;

protected // Can only be used as a sub-rule
DIGIT : '0'..'9' ; // Any character between 0 and 9
INT : (DIGIT)+ ; // One or more digits

WS : ( ' ' | '\t' | '\n' | '\r' ) { $setType(Token.SKIP); } ; // Action: ignore
ANTLR Specifications for Scanners

Rules are names starting with a capital letter.

A character in single quotes matches that character.

LPAREN : ' (' ;

A string in double quotes matches the string

IF : "if" ;

A vertical bar indicates a choice:

OP :  '+' | '-' | '*' | '/' ;
ANTLR Specifications

Question mark makes a clause optional.

```antlr
PERSON : ("wo")? 'm' ('a'|'e') 'n' ;
```

(Matches man, men, woman, and women.)

Double dots indicate a range of characters:

```antlr
DIGIT : '0'..'9';
```

Asterisk and plus match “zero or more,” “one or more.”

```antlr
ID : LETTER (LETTER | DIGIT)* ;
NUMBER : (DIGIT)+ ;
```
Kleene Closure

The asterisk operator (*) is called the Kleene Closure operator after the inventor of regular expressions, Stephen Cole Kleene, who pronounced his last name “CLAY-nee.”

His son Ken writes “As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father.”
Scanner Behavior

All rules (tokens) are considered simultaneously. The longest one that matches wins:

1. Look at the next character in the file.
2. Can the next character be added to any of the tokens under construction?
3. If so, add the character to the token being constructed and go to step 1.
4. Otherwise, return the token.

How to keep track of multiple rules matching simultaneously? Build an automata.
Implementing Scanners Automatically

Regular Expressions (Rules)

↓

Nondeterministic Finite Automata

↓

Subset Construction

Deterministic Finite Automata

↓

Tables
Regular Expressions and NFAs

We are describing tokens with *regular expressions*:

- The symbol $\epsilon$ always matches.
- A symbol from an alphabet, e.g., $a$, matches itself.
- A sequence of two regular expressions e.g., $e_1 e_2$.
  Matches $e_1$ followed by $e_2$.
- An “OR” of two regular expressions e.g., $e_1 \mid e_2$.
  Matches $e_1$ or $e_2$.
- The Kleene closure of a regular expression, e.g., $(e)^*$.
  Matches zero or more instances of $e_1$ in sequence.
Deterministic Finite Automata

A state machine with an initial state

Arcs indicate “consumed” input symbols.

States with double lines are accepting.

If the next token has an arc, follow the arc.

If the next token has no arc and the state is accepting, return the token.

If the next token has no arc and the state is not accepting, syntax error.
Deterministic Finite Automata

ELSE: "else" ;
ELSEIF: "elseif" ;
Deterministic Finite Automata

IF: "if" ;
ID: 'a'..'z' ('a'..'z' | '0'..'9')* ;
NUM: ('0'..'9')+ ;
Nondeterministic Finite Automata

DFAs with $\epsilon$ arcs.

Conceptually, $\epsilon$ arcs denote state equivalence.

$\epsilon$ arcs add the ability to make nondeterministic (schizophrenic) choices.

When an NFA reaches a state with an $\epsilon$ arc, it moves into every destination.

NFAs can be in multiple states at once.
Translating REs into NFAs

$\epsilon$ transitions are not shown for clarity.
RE to NFAs

Building an NFA for the regular expression

\((wo|\epsilon)m(a|e)n\)

produces

![Diagram of NFA]

after simplification. Most \(\epsilon\) arcs disappear.
Subset Construction

How to compute a DFA from an NFA.

Basic idea: each state of the DFA is a *marking* of the NFA.
Subset Construction

An DFA can be exponentially larger than the corresponding NFA.

$n$ states versus $2^n$

Tools often try to strike a balance between the two representations.

ANTLR uses a different technique.
Free-Format Languages

Typical style arising from scanner/parser division

Program text is a series of tokens possibly separated by whitespace and comments, which are both ignored.

- keywords (if while)
- punctuation (, ( +)
- identifiers (foo bar)
- numbers (10 -3.14159e+32)
- strings ("A String")
Free-Format Languages

Java C C++ Algol Pascal

Some deviate a little (e.g., C and C++ have a separate preprocessor)

But not all languages are free-format.
FORTRAN 77

FORTRAN 77 is not free-format. 72-character lines:

```
100 IF (IN .EQ. 'y' .OR. IN .EQ. 'Y' .OR. $ IN .EQ. 't' .OR. IN .EQ. 'T' .OR. IN .EQ. 't') THEN
```

When column 6 is not a space, line is considered part of the previous.

Fixed-length line makes it easy to allocate a one-line buffer.

Makes sense on punch cards.
Python

The Python scripting language groups with indentation

```python
i = 0
while i < 10:
    i = i + 1
    print i  # Prints 1, 2, ..., 10

i = 0
while i < 10:
    i = i + 1
print i  # Just prints 10
```

This is succinct, but can be error-prone.

How do you wrap a conditional around instructions?
Syntax and Langauge Design

Does syntax matter? Yes and no

More important is a language’s semantics—its meaning.

The syntax is aesthetic, but can be a religious issue.

But aesthetics matter to people, and can be critical.

Verbosity does matter: smaller is usually better.

Too small can be a problem: APL is a compact, cryptic language with its own character set (!)

E←A TEST B; L
L←0.5
E←(A×A)+B×B)*L
Syntax and Language Design

Some syntax is error-prone. Classic FORTRAN example:

```
DO 5 I = 1,25 ! Loop header (for i = 1 to 25)
DO 5 I = 1.25 ! Assignment to variable DO5I
```

Trying too hard to reuse existing syntax in C++:

```
vector< vector<int> > foo;
vector<vector<int>> foo; // Syntax error
```

C distinguishes `>` and `>>` as different operators.
Keywords

Keywords look like identifiers in most languages. Scanners do not know context, so keywords must take precedence over identifiers. Too many keywords leaves fewer options for identifiers. Languages such as C++ or Java strive for fewer keywords to avoid "polluting" available identifiers.
Parsing
Parsing

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

\[ 2 \times 3 + 4 \rightarrow \]

\[
    +
   / \\ /
  2 \ 3 \ 4
\]

Goal: discard irrelevant information to make it easier for the next stage.

Parentheses and most other forms of punctuation removed.
Grammars

Most programming languages described using a context-free grammar.

Compared to regular languages, context-free languages add one important thing: recursion.

Recursion allows you to count, e.g., to match pairs of nested parentheses.

Which languages do humans speak? I’d say it’s regular: I do not not not not not not not not not not understand this sentence.
Languages

Regular languages ($t$ is a terminal):

\[ A \rightarrow t_1 \ldots t_n B \]
\[ A \rightarrow t_1 \ldots t_n \]

Context-free languages ($P$ is terminal or a variable):

\[ A \rightarrow P_1 \ldots P_n \]

Context-sensitive languages:

\[ \alpha_1 A \alpha_2 \rightarrow \alpha_1 B \alpha_2 \]

"$B \rightarrow A$ only in the ‘context’ of $\alpha_1 \ldots \alpha_2$"
Issues

Ambiguous grammars
Precedence of operators
Left- versus right-recursive
Top-down vs. bottom-up parsers
Parse Tree vs. Abstract Syntax Tree
Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

\[ 3 - 4 * 2 + 5 \]

with the grammar

\[ e \rightarrow e + e \mid e - e \mid e * e \mid e / e \]

\[
\begin{array}{c}
+ \\
\downarrow \\
- 5 \\
\downarrow \\
3 \\
\downarrow \\
* \\
\downarrow \\
4 2 \\
\end{array}
\quad
\begin{array}{c}
- \\
\downarrow \\
3 + \\
\downarrow \\
* 5 \\
\downarrow \\
4 2 \\
\end{array}
\quad
\begin{array}{c}
* \\
\downarrow \\
\downarrow \\
3 4 2 5 \\
\end{array}
\quad
\begin{array}{c}
- \\
\downarrow \\
3 * \\
\downarrow \\
4 + \\
\downarrow \\
2 5 \\
\end{array}
\quad
\begin{array}{c}
+ \\
\downarrow \\
* 5 \\
\downarrow \\
3 4 \\
\end{array}
\]
Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions

Like you were taught in elementary school:

“My Dear Aunt Sally”

Mnemonic for multiplication and division before addition and subtraction.
Operator Precedence

Defines how “sticky” an operator is.

$1 \times 2 + 3 \times 4$

$\times$ at higher precedence than $+$:
$$(1 \times 2) + (3 \times 4)$$

$+$ at higher precedence than $\times$:
$$1 \times (2 + 3) \times 4$$
# C’s 15 Precedence Levels

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f(r,r,...)</td>
<td>a[i]</td>
<td>p-&gt;m</td>
<td>s.m</td>
<td></td>
</tr>
<tr>
<td>!b</td>
<td>~i</td>
<td>-i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++l</td>
<td>--l</td>
<td>l++</td>
<td>l--</td>
<td></td>
</tr>
<tr>
<td>*p</td>
<td>&amp;l</td>
<td>(type) r</td>
<td>sizeof(t)</td>
<td></td>
</tr>
<tr>
<td>n * o</td>
<td>n / o</td>
<td>i % j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n + o</td>
<td>n - o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i &lt;&lt; j</td>
<td>i &gt;&gt; j</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n &lt; o</td>
<td>n &gt; o</td>
<td>n &lt;= o</td>
<td>n &gt;= o</td>
<td></td>
</tr>
<tr>
<td>r == r</td>
<td>r != r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i &amp; j</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i ^ j</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b &amp;&amp; c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b ? r : r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l = r</td>
<td>l += n</td>
<td>l -= n</td>
<td>l *= n</td>
<td></td>
</tr>
<tr>
<td>l /= n</td>
<td>l %= i</td>
<td>l &amp;= i</td>
<td>l ^= i</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>= i</td>
<td>l &lt;&lt;= i</td>
<td>l &gt;&gt;= i</td>
<td></td>
</tr>
</tbody>
</table>

r1, r2
Associativity

Whether to evaluate left-to-right or right-to-left

Most operators are left-associative

\[ 1 \ - \ 2 \ - \ 3 \ - \ 4 \]

\[
\begin{array}{c}
\frac{\frac{\frac{1}{2}}{3}}{4} \\
\frac{\frac{1}{2}}{3} \\
1 \ 2
\end{array}
\]

left associative

\[
\begin{array}{c}
\frac{\frac{\frac{1}{2}}{3}}{4} \\
\frac{\frac{1}{2}}{3} \\
1 \ 2
\end{array}
\]

right associative
Fixing Ambiguous Grammars

Original ANTLR grammar specification

```plaintext
expr
  :  expr  '+'  expr  
  |  expr  '-'  expr  
  |  expr  '*'  expr  
  |  expr  '/'  expr  
  |  NUMBER  

Ambiguous: no precedence or associativity.
```
Assigning Precedence Levels

Split into multiple rules, one per level

```plaintext
expr : expr '+' expr
    | expr '-' expr
    | term ;

term : term '*' term
    | term '/' term
    | atom ;

atom : NUMBER ;
```

Still ambiguous: associativity not defined
Assigning Associativity

Make one side or the other the next level of precedence

```plaintext
expr : expr '+' term
    | expr '-' term
    | term ;

term : term '*' atom
    | term '/' atom
    | atom ;

atom : NUMBER ;
```
Parsing Context-Free Grammars

There are $O(n^3)$ algorithms for parsing arbitrary CFGs, but most compilers demand $O(n)$ algorithms.

Fortunately, the LL and LR subclasses of CFGs have $O(n)$ parsing algorithms. People use these in practice.
Parsing LL(k) Grammars

LL: Left-to-right, Left-most derivation

k: number of tokens to look ahead

Parsed by top-down, predictive, recursive parsers

Basic idea: look at the next token to predict which production to use

ANTLR builds recursive LL(k) parsers

Almost a direct translation from the grammar.
A Top-Down Parser

stmt : 'if' expr 'then' expr
     | 'while' expr 'do' expr
     | expr ':=' expr ;

eexpr : NUMBER | '(expr ')' ;

AST stmt() –
    switch (next-token) –
    case "if" : match("if"); expr(); match("then"); expr();
    case "while" : match("while"); expr(); match("do"); expr();
    case NUMBER or "(" : expr(); match(":="); expr();
    " " " 
Writing LL(k) Grammars

Cannot have left-recursion

```
expr : expr '++' term | term ;
```

becomes

```
AST expr() –
  switch (next-token) –
  case NUMBER : expr(); /* Infinite Recursion */
```
Writing LL(1) Grammars

Cannot have common prefixes

\[ \text{expr} : \ ID \ ' ( ' \ expr \ ' ) ' \]
\[ \ | \ ID \ ' = ' \ expr \]

becomes

\[
\text{AST expr}() – \\
\text{switch (next-token) –} \\
\text{case ID : match(ID); match(’(‘); expr(); match(’)’);} \\
\text{case ID : match(ID); match(’=’); expr();}
\]
Eliminating Common Prefixes

Consolidate common prefixes:

```plaintext
expr
  : expr '+' term
  | expr '-' term
  | term
  ;

becomes

expr
  : expr ('+' term | '-' term )
  | term
  ;
```
Eliminating Left Recursion

Understand the recursion and add tail rules

\[
\text{expr} \\
\quad : \text{expr} ('+' \text{ term} \mid '-' \text{ term}) \mid \text{term} \\
\]

becomes

\[
\text{expr} : \text{term} \text{ exprt} ; \\
\text{exprt} : '+' \text{ term} \text{ exprt} \mid '-' \text{ term} \text{ exprt} \mid /* \text{nothing} */ \\
\]

Using ANTLR’s EBNF

ANTLR makes this easier since it supports * and -: 

```plaintext
expr : expr '+' term 
   | expr '-' term 
   | term ;
```

becomes 

```plaintext
expr : term ('+' term | '-' term)* ;
```
The Dangling Else Problem

Who owns the else?

```c
if (a) if (b) c(); else d();
```

or if

```c
if a
if b c() d()
```

Grammars are usually ambiguous; manuals give disambiguating rules such as C’s:

As usual the “else” is resolved by connecting an else with the last encountered elseless if.
The Dangling Else Problem

\[
\text{stmt : "if" expr "then" stmt iftail} \\
\quad \mid \text{other-statements ;}
\]

\[
\text{iftail} \\
\quad : \text{"else" stmt} \\
\quad \mid \text{/* nothing */} \\
\]

Problem comes when matching “iftail.”

Normally, an empty choice is taken if the next token is in the “follow set” of the rule. But since “else” can follow an iftail, the decision is ambiguous.
The Dangling Else Problem

ANTLR can resolve this problem by making certain rules “greedy.” If a conditional is marked as greedy, it will take that option even if the “nothing” option would also match:

```plaintext
stmt
  : "if" expr "then" stmt
    ( options {greedy = true;} 
      : "else" stmt
    )?
  | other-statements
```

The Dangling Else Problem

Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

```plaintext
if a < b then a else b fi;
```

“fi” is “if” spelled backwards. The language also uses do–od and case–esac.
Bottom-up Parsers

Regular languages can be matched using finite automata.

Context-free languages can be matched with pushdown automata (have a stack).

Operation of a bottom-up parser:

- Maintain a stack of tokens and rules
- Push each new token onto this stack (“shift”)
- When the top few things on the stack match a rule, replace them (“reduce”)

Used by yacc, bison, and other parser generators.

Parsing more languages, but error recovery harder.
# Bottom-up Parsing

\[
E : T \ ' + ' \ E \ | \ T ; \\
T : \text{int} \ ' * ' \ T \ | \ \text{int} ;
\]

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int + int</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>* int + int</td>
<td>shift</td>
</tr>
<tr>
<td>int *</td>
<td>int + int</td>
<td>shift</td>
</tr>
<tr>
<td>int * int</td>
<td>+ int</td>
<td>reduce ( T : \text{int} )</td>
</tr>
<tr>
<td>int * T</td>
<td>+ int</td>
<td>reduce ( T : \text{int} \ ' * ' \ T )</td>
</tr>
<tr>
<td>T</td>
<td>+ int</td>
<td>shift</td>
</tr>
<tr>
<td>T + int</td>
<td>int</td>
<td>shift</td>
</tr>
<tr>
<td>T + int</td>
<td>reduce ( T : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>T + T</td>
<td>reduce ( E : T )</td>
<td></td>
</tr>
<tr>
<td>T + E</td>
<td>reduce ( E : T \ ' + ' \ E )</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>reduce ( E : T \ ' + ' \ E )</td>
<td></td>
</tr>
</tbody>
</table>
Parsing Techniques

Much theory has been developed about languages and parsing algorithms.

Could easily fill a term.

Fortunately, you don’t need to know all the technical details to build an effective parser using tools.

Just know about tools such as ANTLR, lex, flex, yacc, Bison, JLex, CUP, etc.
Statement separators or terminators?

C uses ; as a statement terminator.

```c
if (a<b) printf("a less");
else {
    printf("b"); printf(" less");
}
```

Pascal uses ; as a statement separator.

```pascal
if a < b then writeln(‘a less’) 
else begin 
    write(‘a’); writeln(‘ less’) 
end
```

Pascal later made a final ; optional.
Summary

Compiler: scanner, parser, AST, IR, assembly
Scanner divides input into tokens
Scanning defined using a regular language
Parser uses rules to recognize phrases and build AST
Context-free grammars used for parsers
Operator precedence and associativity
Top-down and bottom-up parsers