



COMP 181



Lecture 5
Parsing

September 26, 2005



PACT report

- Parallel Architectures and Compilation Techniques
 - Good quality second tier conference
 - Mix of architecture and compiler people
- Talks of interest
 - Compiler for the CELL processor
 - Compiler has to do everything
 - Fetching instructions, predicting branches, moving memory between processing elements
 - Fortress programming language...



Fortress

- Project at Sun Labs in Burlington, MA
Guy Steele and others
- **Goal:** do for Fortran what Java did for C
 - Help high-performance, scientific computing
 - Support natural mathematical expressions
 - ➔ Raise the level of abstraction



Math syntax

- Java:
 - Primitives: $a + b * c$
 - Libraries: `a.add(b.multiply(c))`
- C++
 - Operator overloading: $a + b * c$
 - What about other operations? Set operations?
- Fortress
 - Operator overloading + unicode
 - Libraries: `if ((p ∩ q) ⊆ r) ...`

Fortress



- Extensible
- Explicit parallelism
 - Loops are parallel by default
 - Must declare loop-carried dependences
- Natural math notation
 - What does this mean:
 $3x \sin x + \cos y \log 2y$
 - ➔ Challenge for traditional parsers

Parsing introduction

- Is the following sentence grammatically correct:
The horse ran past the barn fell
- Why?
 - We can use run as a transitive verb
 - I ran the horse past the barn
 - The horse that was ran past the barn
 - Structure
 - Subject: the horse ran past the barn
 - Verb: fell

Hopefully, parsing programming languages won't be this hard!

Parsing



- Parser
 - Reads tokens from the scanner
 - Checks organization of tokens against a *grammar*
 - Construct a *derivation*
 - Derivation drives construction of IR



Study of parsing

- Discovering the derivation of a sentence
 - “Diagramming a sentence” in grade school
 - Formalization:
 - Mathematical model of syntax – a grammar G
 - Algorithm for testing membership in $L(G)$
- Roadmap:
 - Context-free grammars
 - Top-down parsers
 - Ad hoc, often hand-coded, recursive descent parsers*
 - Bottom-up parsers
 - Automatically generated LR parsers*



Specifying syntax with a grammar

- Limitations of regular expressions
 - *Last class*: language of nested parens
 - Need something more powerful
 - Still want formal specification (*for automation*)
- Context-free grammar
 - Set of rules for generating sentences
 - Expressed in Backus-Naur form (BNF)



Context-free grammar

- Example:

#	Production rule
1	$sheepnoise \rightarrow sheepnoise \ baa$
2	$\mid \ baa$

 - “produces” or “generates”
 - Alternative (shorthand)
- Formally: *context-free grammar* is
 - $G = (S, N, T, P)$
 - T : set of terminals (*provided by scanner*)
 - N : set of non-terminals (*represent structure*)
 - $s \in N$: start or goal symbol
 - $P : N \rightarrow (N \cup T)^*$: set of production rules



Language $L(G)$

- Language $L(G)$
 - $L(G)$ is all sentences generated from start symbol
- Generating sentences
 - Use productions as *rewrite rules*
 - Start with goal (or start) symbol – a non-terminal
 - Choose a non-terminal and “expand” it to the right-hand side of one of its productions
 - Only terminal symbols left \rightarrow sentence in $L(G)$
 - Intermediate results known as *sentential forms*



Examples

- Grammar:

#	Production rule
1	$sheepnoise \rightarrow sheepnoise \ baa$
2	$\mid \ baa$
- | Rule | Sentential form |
|------|-----------------|
| - | $sheepnoise$ |
| 2 | baa |
- | Rule | Sentential form |
|------|--------------------------|
| - | $sheepnoise$ |
| 1 | $sheepnoise \ baa$ |
| 1 | $sheepnoise \ baa \ baa$ |
| 2 | $baa \ baa \ baa$ |
- | Rule | Sentential form |
|------|--------------------|
| - | $sheepnoise$ |
| 1 | $sheepnoise \ baa$ |
| 2 | $baa \ baa$ |
- Sheep noises aren't that interesting for compilers...*



Better example

- Language of expressions
 - Numbers and identifiers
 - Allow different binary operators
 - Arbitrary sequences of expressions

#	Production rule
1	$expr \rightarrow expr\ op\ expr$
2	$ \underline{number}$
3	$ \underline{identifier}$
4	$op \rightarrow +$
5	$ -$
6	$ *$
7	$ /$

Expressions can consist of other expressions connected by operators

Numbers and identifiers can fill in any of the positions in the overall expression



Language of expressions

- What's in this language?

#	Production rule
1	$expr \rightarrow expr\ op\ expr$
2	$ \underline{number}$
3	$ \underline{identifier}$
4	$op \rightarrow +$
5	$ -$
6	$ *$
7	$ /$

Rule	Sentential form
-	$expr$
1	$expr\ op\ expr$
3	$\langle id, x \rangle\ op\ expr$
5	$\langle id, x \rangle - expr$
1	$\langle id, x \rangle - expr\ op\ expr$
2	$\langle id, x \rangle - \langle num, 2 \rangle\ op\ expr$
6	$\langle id, x \rangle - \langle num, 2 \rangle * expr$
3	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$

➔ We can build the string " $x - 2 * y$ "
This string is in the language



Derivations

- Using grammars
 - A sequence of rewrites is called a *derivation*
 - Discovering a derivation for a string is *parsing*
 - Different derivations are possible
 - At each step we can choose a non-terminal
 - Rightmost derivation:** always choose right NT
 - Leftmost derivation:** always choose left NT
- (Other "random" derivations – not of interest)



Left vs right derivations

- Two derivations of " $x - 2 * y$ "

Rule	Sentential form
-	$expr$
1	$expr\ op\ expr$
3	$\langle id, x \rangle\ op\ expr$
5	$\langle id, x \rangle - expr$
1	$\langle id, x \rangle - expr\ op\ expr$
2	$\langle id, x \rangle - \langle num, 2 \rangle\ op\ expr$
6	$\langle id, x \rangle - \langle num, 2 \rangle * expr$
3	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$

Left-most derivation

Rule	Sentential form
-	$expr$
1	$expr\ op\ expr$
3	$expr\ op\ \langle id, y \rangle$
6	$expr * \langle id, y \rangle$
1	$expr\ op\ expr * \langle id, y \rangle$
2	$expr\ op\ \langle num, 2 \rangle * \langle id, y \rangle$
5	$expr - \langle num, 2 \rangle * \langle id, y \rangle$
3	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$

Right-most derivation



Derivations and parse trees

- Two different derivations
 - Both are correct
 - Do we care which one we use?
 - Represent derivation as a *parse tree*
 - Leaves are terminal symbols
 - Inner nodes are non-terminals
 - To depict production $\alpha \rightarrow \beta\gamma\delta$ show nodes β, γ, δ as children of α
- ➔ Tree is used to build internal representation

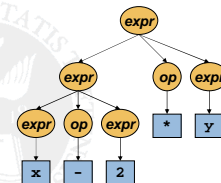


Example (I)

Right-most derivation

Rule	Sentential form
-	$expr$
1	$expr\ op\ expr$
3	$expr\ op\ \langle id, y \rangle$
6	$expr * \langle id, y \rangle$
1	$expr\ op\ expr * \langle id, y \rangle$
2	$expr\ op\ \langle num, 2 \rangle * \langle id, y \rangle$
5	$expr - \langle num, 2 \rangle * \langle id, y \rangle$
3	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$

Parse tree



- Problem:** evaluates as $(x - 2) * y$

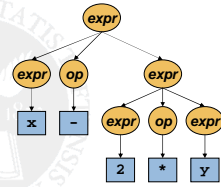


Example (I)

Left-most derivation

Rule	Sentential form
-	<i>expr</i>
1	<i>expr op expr</i>
3	<i><id, x> op expr</i>
5	<i><id, x> - expr</i>
1	<i><id, x> - expr op expr</i>
2	<i><id, x> - <num, 2> op expr</i>
6	<i><id, x> - <num, 2> * expr</i>
3	<i><id, x> - <num, 2> * <id, y></i>

Parse tree



- **Solution:** evaluates as $x - (2 * y)$



Tufts University Computer Science

19

Derivations and precedence

• Problem:

- Two different valid derivations
- Shape of tree implies its meaning
- One captures semantics we want – **precedence**

• Can we express precedence in grammar?

- Notice: operations deeper in tree evaluated first
- **Idea:** add an intermediate production
 - New production isolates different levels of precedence
 - Force higher precedence “deeper” in the grammar



Tufts University Computer Science

20

Adding precedence

• Two levels:

Level 1: lower precedence – higher in the tree

Level 2: higher precedence – deeper in the tree

• Observations:

- Larger: requires more rewriting to reach terminals
- Produces same parse tree under both left and right derivations

#	Production rule
1	<i>expr</i> → <i>expr + term</i>
2	<i>expr - term</i>
3	<i>term</i>
4	<i>term</i> → <i>term * factor</i>
5	<i>term / factor</i>
6	<i>factor</i>
7	<i>factor</i> → <i>number</i>
8	<i>identifier</i>



Tufts University Computer Science

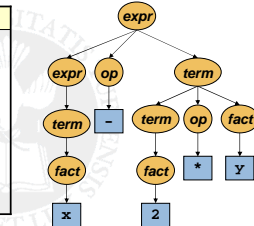
21

Expression example

Right-most derivation

Parse tree

Rule	Sentential form
-	<i>expr</i>
2	<i>expr - term</i>
4	<i>expr - term * factor</i>
8	<i>expr - term * <id, y></i>
6	<i>expr - factor * <id, y></i>
7	<i>expr - <num, 2> * <id, y></i>
3	<i>term - <num, 2> * <id, y></i>
6	<i>factor - <num, 2> * <id, y></i>
8	<i><id, x> - <num, 2> * <id, y></i>



➔ Now right derivation yields $x - (2 * y)$



Tufts University Computer Science

22

Typical patterns

• One or more:

#	Production rule
1	<i>list</i> → <i>element list</i>
2	<i>element</i>

• Zero or more:

#	Production rule
1	<i>list</i> → <i>element list</i>
2	

• Comma separated list:

#	Production rule
1	<i>list</i> → <i>element , list</i>
2	<i>element</i>



Tufts University Computer Science

23

C expressions

```

primary.expression: /* P */ /* 6.3.1 EXTENDED */
| constant
| string.literal.list
| '(' expression ')'
;

postfix.expression: /* P */ /* 6.3.2 CLARIFICATION */
| primary.expression
| postfix.expression '[' expression ']'
| postfix.expression '(' ')'
| postfix.expression '(' argument.expression.list ')'
| postfix.expression * identifier
| postfix.expression ctokARROW identifier
| postfix.expression ctokICR
| postfix.expression ctokDECR
;
    
```



Tufts University Computer Science

24

C expressions

```

argument.expression.list: /* P */ /* 6.3.2 */
  assignment.expression
  | argument.expression.list ',' assignment.expression
  ;

unary.expression: /* P */ /* 6.3.3 */
  postfix.expression
  | ctokICR unary.expression
  | ctokDECR unary.expression
  | unary.operator cast.expression
  | ctokSIZEOF unary.expression
  | ctokSIZEOF '(' type.name ')'
  ;

unary.operator: /* P */ /* 6.3.3 */
  'g' | '*' | '+' | '-' | '~' | '!'
  ;

```



C expressions

```

cast.expression: /* P */ /* 6.3.4 */
  unary.expression
  | '(' type.name ')' cast.expression
  ;

multiplicative.expression: /* P */ /* 6.3.5 */
  cast.expression
  | multiplicative.expression '*' cast.expression
  | multiplicative.expression '/' cast.expression
  | multiplicative.expression '%' cast.expression
  ;

additive.expression: /* P */ /* 6.3.6 */
  multiplicative.expression
  | additive.expression '+' multiplicative.expression
  | additive.expression '-' multiplicative.expression
  ;

```



C expressions

```

shift.expression: /* P */ /* 6.3.7 */
  additive.expression
  | shift.expression ctokLS additive.expression
  | shift.expression ctokRS additive.expression
  ;

relational.expression: /* P */ /* 6.3.8 */
  shift.expression
  | relational.expression '<' shift.expression
  | relational.expression '>' shift.expression
  | relational.expression ctokLE shift.expression
  | relational.expression ctokGE shift.expression
  ;

equality.expression: /* P */ /* 6.3.9 */
  relational.expression
  | equality.expression ctokEQ relational.expression
  | equality.expression ctokNE relational.expression
  ;

```



C expressions

```

AND.expression: /* P */ /* 6.3.10 */
  equality.expression
  | AND.expression '&' equality.expression
  ;

inclusive.OR.expression: /* P */ /* 6.3.12 */
  exclusive.OR.expression
  | inclusive.OR.expression '|' exclusive.OR.expression
  ;

logical.AND.expression: /* P */ /* 6.3.13 */
  inclusive.OR.expression
  | logical.AND.expression ctokANDAND inclusive.OR.expression
  ;

logical.OR.expression: /* P */ /* 6.3.14 */
  logical.AND.expression
  | logical.OR.expression ctokOROR logical.AND.expression
  ;

```



C expressions

```

conditional.expression: /* P */ /* 6.3.15 */
  logical.OR.expression
  | logical.OR.expression '?' expression ':' conditional.expression
  ;

assignment.expression: /* P */ /* 6.3.16 */
  conditional.expression
  | unary.expression assignment.operator assignment.expression
  ;

expression: /* P */ /* 6.3.17 */
  assignment.expression
  | expression ',' assignment.expression
  ;

```



Error productions

- How to provide useful error information?
- **Idea:** add productions for common errors

#	Production rule
1	$expr \rightarrow expr \text{ op } expr$
2	$_ / \text{ number}$
3	$_ \text{ identifier}$
4	$_ / \text{ expr op error}$

- Special "error" token – used in yacc/bison
- Emit message:
"binary operation missing operand"



Ambiguity

- Original example has another problem:

#	Production rule	Rule	Sentential form
1	$expr \rightarrow expr\ op\ expr$	-	$expr$
2	$/\ number$	1	$expr\ op\ expr$
3	$ identifier$	3	$expr\ op\ expr\ op\ expr$
4	$op \rightarrow +$	5	$\langle id, x \rangle\ op\ expr\ op\ expr$
5	$/ -$	2	$\langle id, x \rangle\ -\ expr\ op\ expr$
6	$/ *$	6	$\langle id, x \rangle\ -\ \langle num, 2 \rangle\ * \ expr$
7	$/ /$	3	$\langle id, x \rangle\ -\ \langle num, 2 \rangle\ * \ \langle id, y \rangle$

- Multiple leftmost derivations – hard to automate
- Such a grammar is called **ambiguous**



Ambiguous grammars

- A grammar is ambiguous iff:
 - There are multiple leftmost or multiple rightmost derivations for a single sentential form
 - Note:** leftmost and rightmost derivations may differ, even in an unambiguous grammar
 - Intuitively:**
 - We can choose different non-terminals to expand
 - But each non-terminal should lead to a unique set of terminal symbols
- Classic example: if-then-else ambiguity



If-then-else

- Grammar:

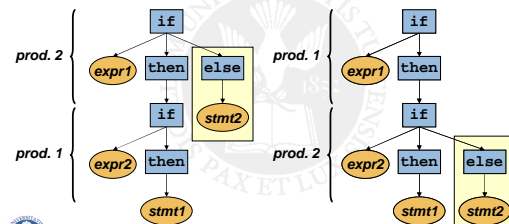
#	Production rule
1	$stmt \rightarrow \text{if } expr \text{ then } stmt$
2	$ \text{if } expr \text{ then } stmt \text{ else } stmt$
3	$ \dots \text{other statements} \dots$

- Problem:** nested if-then-else statements
 - Each one may or may not have else
 - How to match each **else** with **if**



If-then-else ambiguity

- Sentential form with two derivations:
 $\text{if } expr1 \text{ then if } expr2 \text{ then } stmt1 \text{ else } stmt2$



Removing ambiguity

- Restrict the grammar
 - Choose a rule: "else" matches innermost "if"
 - Codify with new productions

#	Production rule
1	$stmt \rightarrow \text{if } expr \text{ then } stmt$
2	$ \text{if } expr \text{ then } \text{withelse } \text{else } stmt$
3	$ \dots \text{other statements} \dots$
4	$\text{withelse} \rightarrow \text{if } expr \text{ then } \text{withelse } \text{else } \text{withelse}$
5	$ \dots \text{other statements} \dots$

- Intuition:** when we have an "else", all preceding nested conditions must have an "else"



Ambiguity

- Ambiguity can take different forms
 - Grammatical ambiguity (*if-then-else problem*)
 - Contextual ambiguity
 - In C: $x * y;$ could follow `typedef int x;`
 - In Fortran: $x = f(y);$ f could be function or array
- Cannot be solved directly in grammar
 - Issues of **type** (later in course)
- Deeper question:
How much can the parser do?



Back to regular languages

- Revisit matching parens problem

#	Production rule
1	$\text{parens} \rightarrow (\text{parens})$
2	$/ ()$
3	$/ \text{parens} (\text{parens})$

- Difference in power:

- Context-free grammars solve problem
Intuitively: allow expansion "in the middle"
- Regular grammars: less powerful

Only allow productions of the form $\alpha \rightarrow \underline{x} \beta$



Regular languages

- Still good for scanning
 - Efficiency (overhead, not asymptotic)
 - Comments, white space

#	Production rule
1	$\text{expr} \rightarrow \text{comm expr op comm expr}$
2	$/ \underline{\text{number}} \text{ comm}$
3	$/ \underline{\text{identifier}} \text{ comm}$
...
8	$\text{comm} \rightarrow /* \text{ text } */$
9	$/$

- Yuck! And hard to get right



Beyond context-free?

- Is there a context-free grammar for:

$$a^n b^n c^n$$

- Must have same number of a's, b's, and c's
- No

Intuitively: need to expand in two places

- Slightly surprising:

$$a^n b^m c^{m+n}$$

is a context-free language

#	Production rule
1	$S \rightarrow \underline{a} S c$
2	$/ B$
3	$B \rightarrow \underline{b} B c$
4	$/ \epsilon$



Big picture

- Scanners
 - Based on regular expressions
 - Efficient for recognizing token types
 - Remove comments, white space
 - Cannot handle complex structure
- Parsers
 - Based on context-free grammars
 - Less efficient
 - More powerful than REs, but still have limitations
- Type and semantic analysis
 - Based on attribute grammars and type systems
 - Handles "context-sensitive" constructs



Roadmap

- So far...

- Context-free grammars, precedence, ambiguity
- Derivation of strings

- Parsing:

- Start with string, discover the derivation
- Two major approaches
 - Top-down – start at the top, work towards terminals
 - Bottom-up – start at terminals, assemble into tree



Next time...

- First programming project
I promise!
- Top-down parsers

