CSE P 501 – Compilers

Parsing & Context-Free Grammars
Hal Perkins
Autumn 2005

Agenda for Today
- Parsing overview
- Context free grammars
- Ambiguous grammars

Parsing
- The syntax of most programming languages can be specified by a context-free grammar (CFG)
- Parsing: Given a grammar $G$ and a sentence $w$ in $L(G)$, traverse the derivation (parse tree) for $w$ in some standard order and do something useful at each node
  - The tree might not be produced explicitly, but the control flow of a parser corresponds to a traversal

"Standard Order"
- For practical reasons we want the parser to be deterministic (no backtracking), and we want to examine the source program from left to right.
  - (i.e., parse the program in linear time in the order it appears in the source file)

Common Orderings
- Top-down
  - Start with the root
  - Traverse the parse tree depth-first, left-to-right (leftmost derivation)
    - LL(k)
- Bottom-up
  - Start at leaves and build up to the root
    - Effectively a rightmost derivation in reverse(!)
  - LR(k) and subsets (LALR(k), SLR(k), etc.)

Old Example

```
program ::= statement | program statement
statement ::= assignStmt | ifStmt
assignStmt ::= id = expr ;
ifStmt ::= if ( expr ) stmt
expr ::= id | int | expr + expr
Id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
G
program := statement | program statement
statement := assignStmt | ifStmt
assignStmt := id = expr ;
ifStmt := if ( expr ) stmt
expr := id | int | expr + expr
Id := a | b | c | i | j | k | n | x | y | z
int := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
W = a = 1 ; if ( a + 1 ) b = 2 ;
```
At each point (node) in the traversal, perform some semantic action:
- Construct nodes of full parse tree (rare)
- Construct abstract syntax tree (common)
- Construct linear, lower-level representation (more common in later parts of a modern compiler)
- Generate target code on the fly (1-pass compiler; not common in production compilers – can't generate very good code in one pass – but great if you need a quick 'n dirty working compiler).

Formally, a grammar $G$ is a tuple $<N, \Sigma, P, S>$ where:
- $N$ a finite set of non-terminal symbols
- $\Sigma$ a finite set of terminal symbols
- $P$ a finite set of productions
- $S$ the start symbol, a distinguished element of $N$
  - If not specified otherwise, this is usually assumed to be the non-terminal on the left of the first production

For $A$ in $N$, $L(A) = \{ w \mid A =>^* w \}$
- If $S$ is the start symbol of grammar $G$, define $L(G) = L(S)$
Reduced Grammars

- Grammar $G$ is reduced iff for every production $A ::= \alpha$ in $G$ there is a derivation $S \Rightarrow^* x A z \Rightarrow^* x \alpha z \Rightarrow^* xyz$
- i.e., no production is useless
- Convention: we will use only reduced grammars

Ambiguity

- Grammar $G$ is unambiguous iff every $w$ in $L(G)$ has a unique leftmost (or rightmost) derivation
- Fact: unique leftmost or unique rightmost implies the other
- A grammar without this property is ambiguous
- Note that other grammars that generate the same language may be unambiguous
- We need unambiguous grammars for parsing

Example: Ambiguous Grammar for Arithmetic Expressions

```
expr ::= expr + expr | expr - expr |
       | expr * expr | expr / expr | int
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- Exercise: show that this is ambiguous
  - How? Show two different leftmost or rightmost derivations for the same string
  - Equivalently: show two different parse trees for the same string

Example (cont)

Give a leftmost derivation of $2 + 3 \times 4$ and show the parse tree

Another example

- Give two different derivations of $5 + 6 + 7$
What’s going on here?

- The grammar has no notion of precedence or associatively
- Solution
  - Create a non-terminal for each level of precedence
  - Isolate the corresponding part of the grammar
  - Force the parser to recognize higher precedence subexpressions first

Classic Expression Grammar

```
expr ::= expr + term | expr – term | term
term ::= term * factor | term / factor | factor
factor ::= int | ( expr )
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

Check: Derive 2 + 3 * 4

Check: Derive 5 + (6 + 7)

Check: Derive 5 + 6 + 7

Note interaction between left- vs right-recursive rules and resulting associativity

Another Classic Example

- Grammar for conditional statements
  ```
  ifStmt ::= if ( cond ) stmt
            | if ( cond ) stmt else stmt
  ```
- Exercise: show that this is ambiguous
  - How?
One Derivation

ifStmt ::= if (cond) stmt | if (cond) stmt else stmt

Another Derivation

ifStmt ::= if (cond) stmt | if (cond) stmt else stmt

Solving if Ambiguity

- Fix the grammar to separate if statements with else clause and if statements with no else
- Done in Java reference grammar
- Adds lots of non-terminals
- Use some ad-hoc rule in parser
  - "else matches closest unpaired if"

Parser Tools and Operators

- Most parser tools can cope with ambiguous grammars
  - Makes life simpler if used with discipline
  - Typically one can specify operator precedence & associativity
  - Allows simpler, ambiguous grammar with fewer nonterminals as basis for generated parser, without creating problems

Parser Tools and Ambiguous Grammars

- Possible rules for resolving other problems
  - Earlier productions in the grammar preferred to later ones
  - Longest match used if there is a choice
- Parser tools normally allow for this
  - But be sure that what the tool does is really what you want

Coming Attractions

- Next topic: LR parsing
  - Continue reading ch. 3