CSE P 501 – Compilers

Parsing & Context-Free Grammars
Hal Perkins
Winter 2016
Administrivia

- Project partner signup: please find a partner and fill out the signup form by noon tomorrow if not done yet (only one form per group, please)
  - Watch for spam from CSE GitLab as repos are set up (save and ignore for now)
- Written HW2 out now, due in a week
- HW1 solution posted in a couple of days
- First part of project – scanner – out later this week, due in two weeks
  - Programming is fairly simple; this is the infrastructure shakedown cruise
Agenda for Today

• Parsing overview
• Context free grammars
• Ambiguous grammars
• Reading: Cooper & Torczon 3.1-3.2
  — Dragon book is also particularly strong on grammars and languages
Syntactic Analysis / Parsing

• Goal: Convert token stream to an abstract syntax tree
• Abstract syntax tree (AST):
  — Captures the structural features of the program
  — Primary data structure for next phases of compilation
• Plan
  — Study how context-free grammars specify syntax
  — Study algorithms for parsing and building ASTs
Context-free Grammars

- The syntax of most programming languages can be specified by a context-free grammar (CGF)
- Compromise between
  - REs: can’t nest or specify recursive structure
  - General grammars: too powerful, undecidable
- Context-free grammars are a sweet spot
  - Powerful enough to describe nesting, recursion
  - Easy to parse; restrictions on general CFGs improve speed
- Not perfect
  - Cannot capture semantics, like “must declare every variable” or “must be int” – requires later semantic pass
  - Can be ambiguous (something we’ll deal with)
Derivations and Parse Trees

• Derivation: a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals

• Parsing: inverse of derivation
  – Given a sequence of terminals (aka tokens) recover (discover) the nonterminals and structure, i.e., the parse tree (concrete syntax)
Old Example

\[ w \rightarrow a = 1 ; \text{if} ( a + 1 ) b = 2 ; \]

UNI CSE P 501 Winter 2016
Parsing

- 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 the parser will correspond 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), recursive-descent

- **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.)
“Something Useful”

- At each point (node) in the traversal, perform some semantic action
  - Construct nodes of full parse tree (rare)
  - Construct abstract syntax tree (AST) (common)
  - Construct linear, lower-level representation (often produced in later phases of production compilers by traversing initial AST)
  - Generate target code on the fly (done in 1-pass compilers; not common in production compilers)
  - Can’t generate great code in one pass, – but useful if you need a quick ‘n dirty working compiler
Context-Free Grammars

- Formally, a grammar $G$ is a tuple $<N, \Sigma, P, S>$ where
  - $N$ is a finite set of non-terminal symbols
  - $\Sigma$ is a finite set of terminal symbols (alphabet)
  - $P$ is a finite set of productions
    - A subset of $N \times (N \cup \Sigma)^*$
  - $S$ is 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
Standard Notations

- $a, b, c$ elements of $\Sigma$
- $w, x, y, z$ elements of $\Sigma^*$
- $A, B, C$ elements of $N$
- $X, Y, Z$ elements of $N \cup \Sigma$
- $\alpha, \beta, \gamma$ elements of $(N \cup \Sigma)^*$

$A \rightarrow \alpha$ or $A ::= \alpha$ if $<A, \alpha> \in P$
Derivation Relations (1)

- \( \alpha A \gamma \Rightarrow \alpha \beta \gamma \) iff \( A ::= \beta \) in \( P \)
  - derives

- \( A \Rightarrow^* \alpha \) if there is a chain of productions starting with \( A \) that generates \( \alpha \)
  - transitive closure
Derivation Relations (2)

• \( wA\gamma \Rightarrow_{\text{lm}} w\beta\gamma \) iff \( A ::= \beta \) in \( P \)
  – derives leftmost

• \( \alpha Aw \Rightarrow_{\text{rm}} \alpha\beta w \) iff \( A ::= \beta \) in \( P \)
  – derives rightmost

• We will only be interested in leftmost and rightmost derivations – not random orderings
Languages

• For A in N, $L(A) = \{ w \mid A \Rightarrow^* w \}$

• If $S$ is the start symbol of grammar $G$, define $L(G) = L(S)$
  – Nonterminal on left of first rule is taken to be the start symbol if one is not specified explicitly
Reduced Grammars

• Grammar $G$ is reduced iff for every production $A ::= \alpha$ in $G$ there is a derivation
  
  $\underbrace{S => x}_S$ $x A z => x \underbrace{\alpha z =>* xyz}_A$  
  — i.e., no production is useless

• Convention: we will use only reduced grammars

✓ — There are algorithms for pruning useless productions from grammars – see a formal language or compiler book for details
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, i.e., ambiguity is a property of grammars, not languages
- We need unambiguous grammars for parsing
Example: Ambiguous Grammar for Arithmetic Expressions

\[ expr ::= expr + expr \mid expr - expr \]
\[ \quad \mid expr \ast expr \mid expr / expr \mid int \]
\[ int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

• 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*4 and show the parse tree
Example (cont)

- Give a different leftmost derivation of 2+3*4 and show the parse tree

\[
expr ::= expr + expr \mid expr - expr \\
  \mid expr \ast expr \mid expr / expr \mid int
\]

\[
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]
Another example

• Give two different derivations of 5+6+7

\[
\begin{align*}
\text{expr} &::= \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \\
&\quad \mid \text{expr} \times \text{expr} \mid \text{expr} / \text{expr} \mid \text{int} \\
\text{int} &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

\[
\begin{align*}
\text{expr} &\quad \text{expr} \\
\text{int} &\quad \text{int} \\
5 &+ (6 + 7) & (5 + 6) + 7 \\
(a - b) - c &\quad a - (b - c)
\end{align*}
\]
What’s going on here?

- The grammar has no notion of precedence or associatively
- Traditional 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
  - Use left- or right-recursion for left- or right-associative operators (non-associative operators are not recursive)
Classic Expression Grammar
(first used in ALGOL 60)

\[ expr ::= expr + term \mid expr - term \mid term \]
\[ term ::= term \times factor \mid term / factor \mid factor \]
\[ factor ::= int \mid ( expr ) \]
\[ int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \]
Check:
Derive $2 + 3 \times 4$

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 $5 + 6 + 7$

- Note interaction between left- vs right-recursive rules and resulting associativity
Check:
Derive $5 + (6 + 7)$
Another Classic Example

• Grammar for conditional statements

\[ stmt ::= \text{if } ( \text{expr} ) \text{ stmt} \]

\[ | \text{if } ( \text{expr} ) \text{ stmt} \text{ else stmt} \]

(This is the “dangling else” problem found in many, many grammars for languages beginning with Algol 60)

— Exercise: show that this is ambiguous

• How?
One Derivation

\[ stmt ::= \text{if ( expr ) stmt} \]
\[ \quad \quad \quad \quad | \text{if ( expr ) stmt else stmt} \]
Another Derivation

\[ stmt ::= \text{if ( expr ) stmt} \]
\[ \quad \text{if ( expr ) stmt else stmt} \]
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
• or, Change the language
  — But it’d better be ok to do this – you need to “own” the language or get permission from owner
• or, Use some ad-hoc rule in the parser
  — “else matches closest unpaired if”
Resolving Ambiguity with Grammar (1)

Stmt ::= MatchedStmt | UnmatchedStmt
MatchedStmt ::= ... | if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= ... | if ( Expr ) Stmt |
if ( Expr ) MatchedStmt else UnmatchedStmt

— formal, no additional rules beyond syntax
— can be more obscure than original grammar
Check

Stmt ::= MatchedStmt | UnmatchedStmt
MatchedStmt ::= ... |
             if (Expr) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if (Expr) Stmt |
                  if (Expr) MatchedStmt else UnmatchedStmt

if (expr) if (expr) stmt else stmt
Resolving Ambiguity with Grammar (2)

- If you can (re-)design the language, just avoid the problem entirely

\[
\text{Stmt ::= ... | if Expr then Stmt end | if Expr then Stmt else Stmt end}
\]

- formal, clear, elegant
- allows sequence of Stmts in then and else branches, no \{\,\} needed
- extra end required for every if
  (But maybe this is a good idea anyway?)
Parser Tools and Operators

• Most parser tools can cope with ambiguous grammars
  - Makes life simpler if used with discipline
• Usually can specify precedence & associativity
  - Allows simpler, ambiguous grammar with fewer nonterminals as basis for parser – let the tool handle the details (but only when it makes sense)
  • (i.e., expr ::= expr+expr | expr*expr | ... with assoc. & precedence declarations can be the best solution)
Parser Tools and Ambiguous Grammars

- Possible rules for resolving other problems:
  - Earlier productions in the grammar preferred to later ones (some danger here if grammar changes)
  - Longest match used if there is a choice (good solution for dangling if)

- Parser tools normally allow for this
  - But be sure that what the tool does is really what you want
    - And that it’s part of the tool spec, so that v2 won’t do something different (that you don’t want!)
Coming Attractions

• Next topic: LR parsing
  — Continue reading ch. 3