Recall: Programming Language Specs

- Syntax of every significant programming language is specified by a formal grammar
  » BNF or some variation there on
- As language engineering has developed, formal methods have improved for defining useful grammars and tools for processing them

Recall: Productions

- The rules of a grammar are called *productions*
- Rules contain
  » Nonterminal symbols: grammar variables (*program*, *statement*, *id*, etc.)
  » Terminal symbols: concrete syntax that appears in programs: a, b, c, 0, 1, if, (, ...
- Meaning of nonterminal → <sequence of terminals and nonterminals>
  In a derivation, an instance of nonterminal can be replaced by the sequence of terminals and nonterminals on the right of the production
- Often, there are two or more productions for a single nonterminal – can use either at different times

Grammar for fm, a little language

1. program → movie name { movieBody } EOF
2. movieBody → prologBlock pageBlocks | pageBlocks
3. prologBlock → prolog { prologStatements }
4. prologStatements → prologStatement | prologStatements prologStatement
5. prologStatement → variableDeclaration
6. variableDeclaration → id : type(); | id : type(exprList);
7. pageBlocks → pageBlock | pageBlocks pageBlock
8. pageBlock → show ( integer ) { pageStatements }
9. pageStatements → pageStatement | pageStatements pageStatement
10. pageStatement → { pageStatements } | methodCall; | id = expr;
11. exprList → expr | exprList , expr
12. if ( boolExpr ) pageStatement | if ( boolExpr ) pageStatement else pageStatement
13. boolExpr → relExpr | ! ( relExpr )
14. relExpr → expr == expr | expr > expr | expr < expr
15. type → id
16. type → id | methodCall | id
17. factor → integer | real | ( expr ) | id | methodCall
18. methodCall → id() | id(exprList) | id.id() | id.id(exprList)
19. exprList → expr | exprList , expr
20. methodCall → id() | id(exprList) | id.id() | id.id(exprList)
21. boolExpr → relExpr | ! ( relExpr )
22. relExpr → expr == expr | expr > expr | expr < expr
23. type → id
Grammar for Java, a big language

  » Entire document
    • 500+ pages
    • Grammar productions with explanatory text
  » Chapter 18, Syntax
    • 8 pages of grammar productions, presented in "BNF-style"

Java grammar extract

Type:
  Identifier | . Identifier | BracketsOpt
BasicType
StatementExpression:
  Expression
ConstantExpression:
  Expression
Expression1:
  Expression2 [Expression1Rest]
Expression1Rest:
  [ ? Expression : Expression1]
Expression2:
  Expression3 [Expression2Rest]
Expression2Rest:
  {Infixop Expression3}
Expression3 instanceof Type

Recall Parsing

- Parsing: reconstruct the derivation (syntactic structure) of a program
- In principle, a single recognizer could work directly from the concrete, character-by-character grammar
- In real compilers the recognizer is split into two phases
  » Scanner: translate input characters to tokens
  » Parser: read token stream and reconstruct the derivation

Parsing

- The syntax of most programming languages can be specified by a context-free grammar
- 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*.

  » in other words, parse the program in linear time in the order it appears in the source file

“Something Useful”

- 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
    - relatively simple to program by hand
    - not common in production compilers because can’t generate very good code in one pass

Common Orderings

- Top-down
  » Start with the root
  » Traverse the parse tree depth-first

- Bottom-up
  » Start at leaves and build up to the root
Context-Free Grammars

- 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
    - A subset of $N \times (N \cup \Sigma)^*$
  - $S$ 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

Standard Notations

- $a, b, c$ elements of $\Sigma$ (terminals)
- $w, x, y, z$ elements of $\Sigma^*$ (strings of terminals)
- $A, B, C$ elements of $N$ (non-terminals)
- $X, Y, Z$ elements of $N \cup \Sigma$ (grammar symbols)
- $\alpha, \beta, \gamma$ elements of $(N \cup \Sigma)^*$ (strings of symbols)
- $A \rightarrow \alpha$ (or $A ::= \alpha$) if $<A, \alpha>$ in $P$
  - "non-terminal A can take the form alpha"

Derivation Relations

- $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ iff $A \rightarrow \beta$ in $P$
  - " $\Rightarrow$ " is read "derives"
- $A \Rightarrow^* w$ if there is a chain of productions starting with $A$ that generates $w$
  - "Non-terminal A derives the string of terminals w"
  - You can get from $A$ to $w$ using a series of productions
    - for example, if $S$ is the start symbol "program" and $w$ is the actual source code, then $S \Rightarrow^* w$ says that $w$ is a valid program (i.e., it compiles)

Languages

- For $A$ in $N$, $L(A) = \{ w \mid A \Rightarrow^* w \}$
  - for any non-terminal $A$ defined for a grammar, the language generated by $A$ is the set of strings $w$ that can be derived from $A$ using the productions
- If $S$ is the start symbol of grammar $G$, define $L(G) = L(S)$
  - The language derived by $G$ is the language derived by the start symbol $S$
Reduced Grammars

- Grammar \( G \) is reduced iff for every production \( A \rightarrow \alpha \) 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 derivation
- 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

Ambiguous Grammar for Expressions

\[
\begin{align*}
expr & \rightarrow expr + expr \mid expr - expr \\
 & \mid expr \ast expr \mid expr / expr \mid int \\
int & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

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

Example Derivation

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

Another Derivation

$$\text{expr} \rightarrow \text{expr} + \text{expr} | \text{expr} - \text{expr}$$
$$\text{expr} \rightarrow \text{expr} * \text{expr} | \text{expr} / \text{expr} | \text{int}$$
$$\text{int} \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$

Another Example

Give two different derivations of $5+6+7$

What’s going on here?

• The grammar has no notion of precedence or associativity
• 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

$$\text{expr} \rightarrow \text{expr} + \text{term} | \text{expr} - \text{term} | \text{term}$$
$$\text{term} \rightarrow \text{term} * \text{factor} | \text{term} / \text{factor} | \text{factor}$$
$$\text{factor} \rightarrow \text{int} | ( \text{expr} )$$
$$\text{int} \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7$$
Another Classic Example

- Grammar for conditional statements
  
  \[\text{ifStmt} \rightarrow \text{if} (\text{cond}) \text{stmt} \]
  
  \[\text{if} (\text{cond}) \text{stmt} \quad \text{else} \text{stmt}\]

  » Exercise: show that this is ambiguous
  
  • How?
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”