Grammar

CSE 413, Autumn 2005

Programming Languages

http://www.cs.washington.edu/education/courses/413/05au/

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*
11. *variableDeclaration* → *id* **:** *type***();** | *id* **:** *type***(***exprList***);**
12. *pageBlocks* → *pageBlock* | *pageBlocks pageBlock*
13. *pageBlock* → **show (** *integer* **) {** *pageStatements* **}**
14. *pageStatements* → *pageStatement* | *pageStatements pageStatement*
15. *pageStatement* → **{** *pageStatements* **}** | *methodCall;* | *id* **=** *expr***;**
    | **if (***boolExpr***)** *pageStatement* | **if (***boolExpr***)** *pageStatement* **else** *pageStatement*
16. *expr* → *term* | *expr* **+** *term* | *expr* **-** *term*
17. *term* → *factor* | *term* ***** *factor* | *term* / *factor*
18. *factor* → *integer* | *real* | **(** *expr* **)** | *id* | *methodCall*
19. *methodCall* → *id*() | *id*(*exprList*) | *id***.***id* **( )** | *id***.***id***(***exprList***)**
20. *exprList* → *expr* | *exprList* **,** *expr*
21. *boolExpr* → *relExpr* | **!** (*relExpr*)
22. *relExpr* → *expr* **==** *expr* | *expr* **>** *expr* | *expr* **<** *expr*
23. *type* → *id*

Grammar for Java, a big language

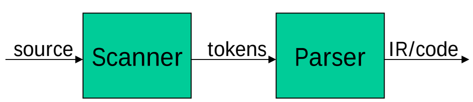
- The Java™ Language Specification, 2nd Edition
  - » *Entire document*
    - *500+ pages*
    - *Grammar productions with explanatory text*
  - » *Chapter 18, Syntax*
    - *8 pages of grammar productions, presented in "BNF-style"*

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



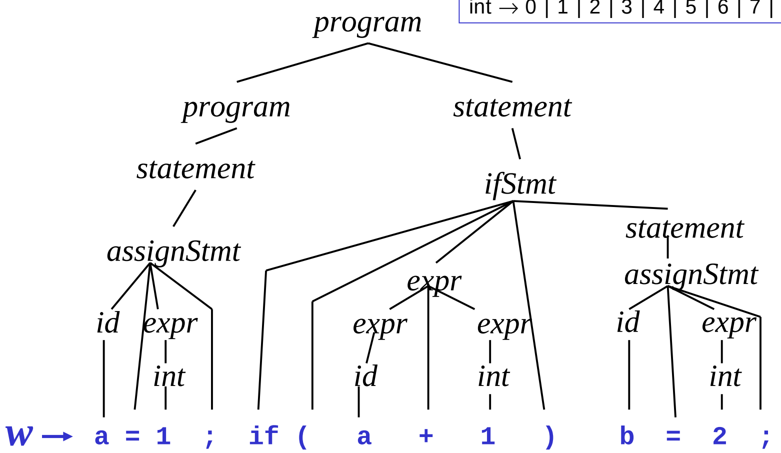
Parsing

- 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

Parse Tree Example

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



"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

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

"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

Context-Free Grammars

- Formally, a grammar *G* is a tuple $<N,\Sigma,P,S>$ where
  - » *N* a finite set of non-terminal symbols
  - » Σ 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 Σ	*terminals*
w, x, y, z	elements of Σ*	*strings of terminals*
A, B, C	elements of *N*	*non-terminals*
X, Y, Z	elements of $N \cup \Sigma$	*grammar symbols*
α, β, γ	elements of $(N \cup \Sigma)^*$	*strings of symbols*
A→α (or A ::= α) if <A, α > in *P*		*productions*

"non-terminal A can take the form α"

Derivation Relations

- $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ iff $A \rightarrow \beta$ in *P*
  - » " ⇒ " is read "derives"
- A ⇒* 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 ⇒* w says that w is a valid program (ie, 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 → α in *G* there is a derivation

  S =>* x A z => x α z =>* 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

expr → *expr* + *expr* | *expr* - *expr*
| *expr* * *expr* | *expr* / *expr* | *int*

int → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- 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

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

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

Another Derivation

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

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

Another Example

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

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

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

Derive 2 + 3 * 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

Derive 5 + 6 + 7

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

Derive 5 + (6 + 7)

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

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*

if (*cond*) if (*cond*) *stmt* else *stmt*

Another Derivation

ifStmt → if (*cond*) *stmt*
| if (*cond*) *stmt* else *stmt*

if (*cond*) 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"