CSE 401 – Compilers

Lecture 5: Parsing & Context-Free Grammars

Michael Ringenburg

Winter 2013

Announcements/ Reminders

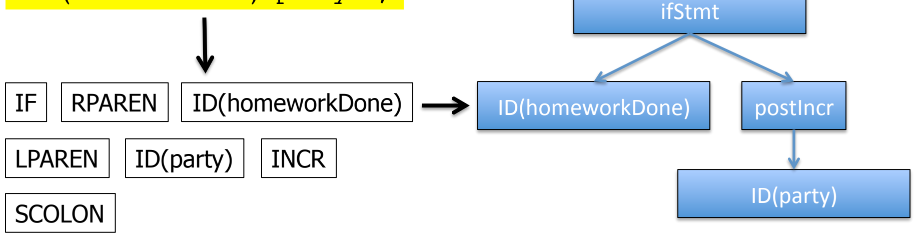
- Homework 1 is due on Friday
  - There have been some good questions posted on the class discussion board – it would be worth your time to check them out.
  - Homework 1 *only* covers material from the scanning lectures. Don't worry about things we'll cover today, like ambiguous grammars.
- No class or office hours Monday (MLK day)
- Class email list
  - Everyone receiving messages? Last one sent Tuesday AM (Subject: Project Teams). Got some bounce notifications.
- Sections tomorrow – AA moved from Savery to MEB 238

Agenda for Today

- Parsing overview
- Context free grammars
- Ambiguous grammars
- Reading: Cooper & Torczon 3.1-3.2

Parsing

```
if (homeworkDone) party++;
```



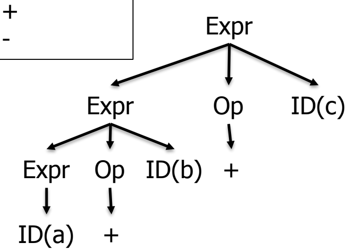
- We have: a scanner that generates a token stream
- We want an abstract syntax tree (AST)
  - A data structure that encodes the *meaning* of the program, and captures its structural features (loops, conditionals, etc.)
  - Primary data structure for next phases of compilation

How is this done?

- A grammar specifies the syntax of a language.
- Parsing algorithms build *parse trees* based on a grammar and a stream of tokens.
  - Parse trees represent how a string can be derived from a grammar, and *encode meaning*.
    - E.g., add *a* and *b*, then add result to *c*.
  - Can build AST by traversing parse tree (parsers may do this implicitly).

```
Expr → Expr Op ID
     | ID
Op   → +
     | -
```

a + b + c



Derivations vs. Parsing

- 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 (a.k.a. tokens) want to recover the nonterminals and structure (i.e., given string of terminals, find a derivation that generates them)
- Can represent derivation as a parse tree

Example Derivation

G

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
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

program

↓ ?

a = 1 ;

Example Derivation

G

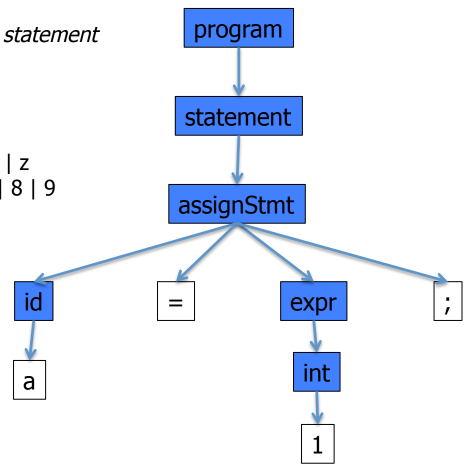
program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

program
statement
assignStmt
id = *expr* ;
a = *expr* ;
a = *int* ;
a = 1 ;

Example Derivation

G

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
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

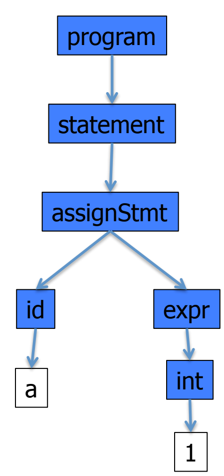


program
statement
assignStmt
id = *expr* ;
a = *expr* ;
a = *int* ;
a = 1 ;

Example Derivation

G

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
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



program
statement
assignStmt
id = *expr* ;
a = *expr* ;
a = *int* ;
a = 1 ;

(Sometimes drawn without redundant terminals)

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 parse tree might not be produced explicitly, but the control flow of the parser corresponds to a traversal
  - For example, can generate the AST directly based on this traversal, without ever actually building the intermediate parse tree.

Parsing

- For efficiency we want the parser to avoid *backtracking* (don't make a wrong guess).
  - Can be done if you allow parser to "lookahead", and specify the grammar properly.
  - Keeps time linear in the size of source code.
- Also want to examine the source program from *left to right.*
  - Parse the program in the order tokens are returned from the scanner.

Common Orderings

- Top-down
  - Start with the root (*start symbol* of grammar G)
  - Traverse the parse tree depth-first, left-to-right
  - Equivalent to a leftmost derivation parse tree (expanding the leftmost nonterminal at every step).
  - LL(k) parsers
- Bottom-up
  - Start at leaves (string of terminals) and build up to the root
  - Ends up generated a rightmost derivation, upside down
    - Referred to as a "Reverse Rightmost Derivation"
  - LR(k) and subsets (LALR(k), SLR(k), etc.)

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/
leftmost

E

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/ leftmost

E
E + *F*

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/ leftmost

E
E + *F*
E + *F* + *F*

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/ leftmost

E
E + *F*
E + *F* + *F*
F + *F* + *F*

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/ leftmost

E
E + *F*
E + *F* + *F*
F + *F* + *F*
a + *F* + *F*

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/
leftmost

E
E + *F*
E + *F* + *F*
F + *F* + *F*
a + *F* + *F*
a + b + *F*

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/
leftmost

E
E + *F*
E + *F* + *F*
F + *F* + *F*
a + *F* + *F*
a + b + *F*
a + b + c

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/ leftmost	Bottom up/ reverse rightmost
E	
E + *F*	
E + *F* + *F*	
F + *F* + *F*	
a + *F* + *F*	
a + b + *F*	
a + b + c	a + b + c

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```


Top down/ leftmost	Bottom up/ reverse rightmost
E	
E + *F*	
E + *F* + *F*	
F + *F* + *F*	
a + *F* + *F*	
a + b + *F*	*F* + b + c
a + b + c	a + b + c

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/ leftmost

E
E + *F*
E + *F* + *F*
F + *F* + *F*
a + *F* + *F*
a + b + *F*
a + b + c

Bottom up/ reverse rightmost

E + b + c
F + b + c
a + b + c

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/ leftmost

E
E + *F*
E + *F* + *F*
F + *F* + *F*
a + *F* + *F*
a + b + *F*
a + b + c

Bottom up/ reverse rightmost

E + *F* + c
E + b + c
F + b + c
a + b + c

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/ leftmost	Bottom up/ reverse rightmost
E	
E + *F*	
E + *F* + *F*	*E* + c
F + *F* + *F*	*E* + *F* + c
a + *F* + *F*	*E* + b + c
a + b + *F*	*F* + b + c
a + b + c	a + b + c

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

a + b + c

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

Top down/ leftmost	Bottom up/ reverse rightmost
E	
E + *F*	*E* + *F*
E + *F* + *F*	*E* + c
F + *F* + *F*	*E* + *F* + c
a + *F* + *F*	*E* + b + c
a + b + *F*	*F* + b + c
a + b + c	a + b + c

Top Down/Leftmost vs. Bottom Up/Reverse Rightmost

```
E ::= E + F
    | F
F ::= a
    | b
    | c
```

a + b + c

Top down/ leftmost	Bottom up/ reverse rightmost
E	*E*
E + *F*	*E* + *F*
E + *F* + *F*	*E* + c
F + *F* + *F*	*E* + *F* + c
a + *F* + *F*	*E* + b + c
a + b + *F*	*F* + b + c
a + b + c	a + b + c

"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 by traversing initial AST in optimization phases of production compilers)
  - Generate target code on the fly (used in 1-pass compiler; not common in production compilers)
    - Can't generate great code in one pass, – but useful if you need a quick 'n dirty working compiler

Specifying Grammar

- Why not just use a Regular Expression?
  - Can't express recursive structure – try creating an RE for balanced parenthesis
    - () just does one.
    - (()) just does two.
    - (*)* - Doesn't guarantee balance.
    - Need something like parens = (parens), but this is recursive and thus not a regular expression.
  - Fundamental problem: REs can't "count" arbitrarily.
  - Makes sense – DFAs (which can encode any RE) can't count either, because they only have finite states, and no memory (beyond state).

Context-free Grammars

- So instead, programming languages are typically specified via a context-free grammar (CFG)
  - CFGs can be recognized by push down automata, which are essentially FAs plus a stack (makes counting possible)
- Context-free grammars are a sweet spot
  - Powerful enough to describe nesting, recursion
  - But easy to parse, unlike some more general grammars
- Not perfect
  - Cannot capture semantics, as in "variable must be declared"
  - Can be *ambiguous* – i.e., multiple ways to derive a string, which may lead to different "meanings" (e.g., order of operation)

Context-Free Grammars

- Formally, a grammar *G* is a tuple <*N*,Σ,*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 $N \times (N \cup \Sigma)^*$, i.e., a non-terminal right-hand side, and zero or more terminals and non-terminals on the left-hand side.
  - *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* (terminal strings)
- A, B, C elements of *N* (nonterminals)
- X, Y, Z elements of $N \cup \Sigma$ (terminals or nonterms)
- α, β, γ elements of $(N \cup \Sigma)^*$ (term/nonterm strings)
- $A \rightarrow \alpha$ or A ::= α if <A, α> in *P* (productions)

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 α
  - transitive closure

Derivation Relations (2)

- $w\ A\ \gamma \Rightarrow_{lm} w\ \beta\ \gamma$ iff $A ::= \beta$ in *P*
  - derives leftmost
- $\alpha\ A\ w \Rightarrow_{rm} \alpha\ \beta\ w$ 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 | A =>* 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 ::= α in *G* there is a derivation

  S =>* x A z => x α z =>* xyz
  - i.e., no production is useless
  - E.g., S ::= hello, Y::=goodbye is *not* reduced (no derivation starting at S will ever use the Y production)
- 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
  - The derivation determines the shape of the parse tree/ abstract syntax tree, which in turn determines meaning.

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

Exercise (cont)

- Give two different leftmost derivations of 2+3*4 and show the parse tree

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

Another exercise

- Give two different rightmost derivations of 5+6+7

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

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
    - E.g., *E* ::= *E* + *F* for left associative addition

Classic Unambiguous 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 | 8 | 9

Check: 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 | 8 | 9

Check: Derive 5 + 6 + 7

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

expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | (*expr*)
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Check: 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 | 8 | 9

Another Classic Example

- Grammar for conditional statements

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

  – Exercise: show that this is ambiguous
    - How?

Another Classic Example

- Grammar for conditional statements

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

  – Exercise: show that this is ambiguous
    - How?
    - Hint: Consider

      if (Cond1) if (Cond2) Stmt1 else Stmt2

Derive if(c1) if(c2) s1 else s2

stmt ::= 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
- or, Change the language
  - But it'd better be ok to do this
- 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

- Prevents if-without-else inside then clause of if-then-else, forcing else to match closest if. But, can still generate exact same language (try it!)
- formal, no additional rules beyond syntax

Check: if (*c1*) if (*c2*) *stmt* else *stmt*

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

Resolving Ambiguity with Grammar (2)

- If you can (re-)design the language, can avoid the problem entirely, e.g., create an **end** to match closest **if**

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? These ambiguities can lead to programmer bugs ...)

Parser Tools and Operators

- Most parser tools can cope with ambiguous grammars
  - Makes life simpler if you're careful
- 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