CSE 401 – Compilers

Parsing & Context-Free Grammars

Hal Perkins

Winter 2015

Administrivia

- New today: project partner signup. Please use the link on the course home page to fill out a form with your and your partner's names and cse netids
  - Only one form per group, please
  - By the end of the weekend, if possible
- New: Please log in to CSE gitlab if you've never done so – we need that to set up your group repo
- Old: last chance to fill in office hour doodle
- Old: have you posted a followup on the discussion board yet?

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; but also allow restrictions for speed
- Not perfect
  - Cannot capture semantics, like "must declare every variable" or "must be `int`" – requires later semantic pass
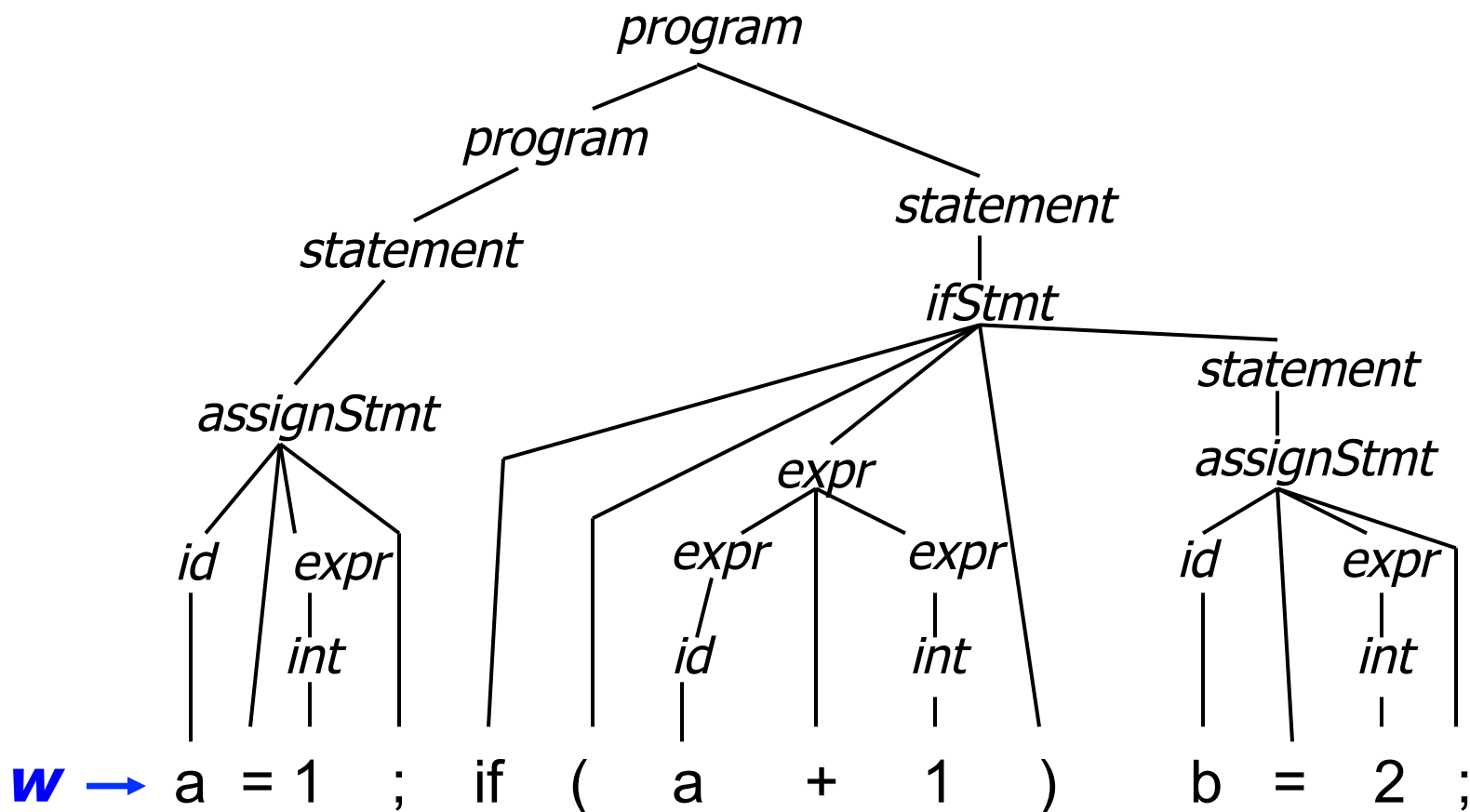 - Can be ambiguous

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) want to recover (discover) the nonterminals and structure, i.e., the parse (concrete syntax) tree

Old Example

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
              program                    statement
         statement                         ifStmt
    assignStmt                                    statement
                                                  assignStmt
   id    expr          expr    expr              id    expr
         int           id      int                     int
w →  a = 1  ;   if  (  a   +   1   )         b  =  2  ;
```

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 by traversing initial AST in later 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

Context-Free Grammars

- Formally, a *grammar* G is a tuple $<N,\Sigma,P,S>$ where
  - *N* is a finite set of *non-terminal* symbols
  - Σ 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 Σ

w, x, y, z elements of Σ^*

A, B, C elements of N

X, Y, Z elements of $N \cup \Sigma$

α, β, γ elements of $(N \cup \Sigma)^*$

$A \rightarrow \alpha$ or $A ::= \alpha$ if $<A, \alpha>$ in P

Derivation Relations (1)

- $\alpha\ A\ \gamma => \alpha\ \beta\ \gamma$ iff $A ::= \beta$ in *P*
  - derives

- $A =>^* \alpha$ 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

- 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

- 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)

expr ::= *expr + expr | expr - expr*
        | expr * expr | expr / expr | int

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

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

Example (cont)

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

- Give a different leftmost 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 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* | *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

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

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

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

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

Another Classic Example

- Grammar for conditional statements

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

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

  - Exercise: show that this is ambiguous
    - How?

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

One Derivation

if ($cond$)   if ($cond$) $stmt$ else $stmt$

Another Derivation

$stmt ::=$ 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
- 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

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

Resolving Ambiguity with Grammar (2)

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

    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

- 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
    - 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