CSE401: Parsing (A)

David Notkin
Autumn 2000

Objectives: parsing lectures

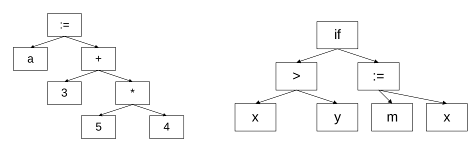
- Understand the theory and practice of parsing
- Describe the underlying language theory of parsing (CFGs, etc.)
- Understand and be able to perform top-down parsing
- Understand bottom-up parsing
- Today's focus: grammars and ambiguity

Parsing

- AST
  - Captures hierarchical structure of the source program
  - It is the primary representation of the program used by the rest of the compiler
    - It gets augmented and annotated, but the basic structure of the AST is used throughout

Parsing: two jobs

- Ensure that the program is syntactically correct
  - `a := 3 + 5 * 4;` vs. `a := 3 * / 4;`
  - `if x > y then m := x;` vs. `if x < y else m := x;`
- Put the sequence of tokens into the AST structure



Context-free grammars (CFGs)

- For lexing, we used regular expressions as the basic underlying notation
- For parsing, we use context-free grammars in much the same way
  - Regular expressions are not powerful enough
    - Intuitively, they can't handle balanced nesting (a^nb^n)
  - And more general grammars are more powerful than we need
    - Well, we could use more power, but instead we delay some checking to semantic analysis instead of making all the analysis based on CFGs

CFG terminology

- *Terminals*: the alphabet (e.g., set of legal tokens)
- *Nonterminals*: symbols defined in terms of terminals and nonterminals
- *Production*: rule that defines a nonterminal in terms of a finite sequence of terminals and nonterminals
- *Start symbol*: root symbol defining the language
-
```
Program ::= Stmt
Stmt    ::= if Expr then Stmt else Stmt end
Stmt    ::= while Expr do Stmt end
```

EBNF description of PL/0 syntax

```
Program   ::= module Id ; Block Id .
Block     ::= DeclList begin StmtList end
DeclList  ::= { Decl ; }
Decl      ::= ConstDecl | ProcDecl | VarDecl
ConstDecl ::= const ConstDeclItem { , ConstDeclItem }
ConstDeclItem ::= Id : Type = ConstExpr
ConstExpr ::= Id | Integer
VarDecl   ::= var VarDeclItem { , VarDeclItem }
VarDeclItem ::= Id : Type
```


EBNF description of PL/0 syntax

```
ProcDecl  ::= procedure Id (
              [ FormalDecl {, FormalDecl} ] ) ;
              Block Id
FormalDecl::= Id : Type
Type      ::= int
StmtList  ::= { Stmt ; }
Stmt      ::= CallStmt | AssignStmt | OutStmt |
              IfStmt | WhileStmt
CallStmt  ::= Id ( [ Exprs ] )
AssignStmt::= Lvalue := Expr
Lvalue    ::= Id
```


EBNF description of PL/0 syntax

```
OutStmt   ::= output := Expr
IfStmt    ::= if Test then StmtList end
WhileStmt ::= while Test do StmtList end
Test      ::= odd Sum | Sum Relop Sum
Relop     ::= <= | <> | < | >= | > | =
Exprs     ::= Expr {, Expr }
Expr      ::= Sum
Sum       ::= Term { (+ | -) Term }
Term      ::= Factor { (* | /) Factor }
Factor    ::= - Factor | LValue | Integer |
              input | ( Expr )
```


Produce a syntax tree for squares.0

in groups, 5 minutes

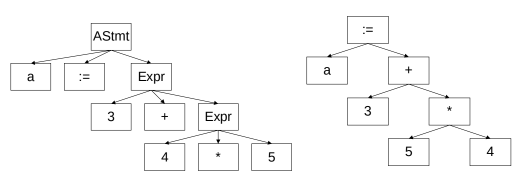
```
module main;
  var x:int, squareret:int;
  procedure square(n:int);
  begin
    squareret := n * n;
  end square;
begin
  x := input;
  while x <> 0 do
    square(x);
    output := squareret;
    x := input;
  end;
end main.
```


Derivations and parsing

- Derivation
  - A sequence of expansion steps,
  - Beginning with the start symbol,
  - Leading to a string of terminals
- Parsing: inverse of derivation
  - Given a target string of terminals,
  - Recover nonterminals representing structure

Parse trees

- We can represent derivations and parses as a *parse tree*
  - Concrete syntax tree
  - Abstract syntax tree



An example expression grammar

```
E   ::= E Op E | - E | ( E ) | id
Op  ::= + | - | * | /
```

- In groups, use this grammar and quickly find parse trees for
  A. `3 * 5`
  B. `3 + 4 * 5`

Ambiguity

- Some grammars are *ambiguous*
  - Multiple different parse trees with the same final string
  - (Some *languages* are ambiguous, with no possible non-ambiguous grammar; but we shy away from them)
- Since the structure of the parse tree captures some of the meaning of a program
  - Ambiguity is bad since it implies multiple possible meanings for the same program
- Consider the example on the previous slide

Another famous ambiguity: *dangling else*

```
Stmt  ::= … |
          if Expr then Stmt |
          if Expr then Stmt else Stmt
```

- **`if`** `e1` **`then if`** `e2` **`then`** `s1` **`else`** `s2`
- To which `then` does the `else` belong?
  - The compiler isn't going to be confused
  - However, if the compiler chooses a meaning different from what the programmer intended, it could get ugly
- Any ideas for overcoming this problem?

Resolving the ambiguity: #1

- Add a meta-rule
  - For instance, "`else` associates with the closest previous `if`"
- This works and keeps the original grammar intact
- But it's ad hoc and informal

Resolving the ambiguity: #2

- Rewrite the grammar to resolve the ambiguity explicitly

```
Stmt          ::= MatchedStmt | UnmatchedStmt
MatchedStmt   ::= … |
                  if Expr then MatchedStmt
                          else MatchedStmt
UnmatchedStmt ::= if Expr then Stmt |
                  if Expr then MatchStmt
                          else UnmatchedStmt
```

- Formal, no additional meta-rules
- Somewhat more obscure grammar

Resolving the ambiguity: #3

- Redesign the programming language to remove the ambiguity

```
Stmt   ::= if Expr then Stmt end |
           if Expr then Stmt else Stmt end
```

- Formal, clear, elegant
- Allows `StmtList` in `then` and `else` branch, without adding `begin/end`
- Extra `end` required for every `if` statement

What about that expression grammar?

- How to resolve its ambiguity?
- Option #1: add meta-rules for precedence and associativity
- Option #2: modify the grammar to explicitly resolve the ambiguity

Option #2: strategy

- Create a nonterminal for each precedence level
- `Expr` is the lowest precedence nonterminal
  - Each nonterminal can be rewritten with higher precedence operator
  - Highest precedence operator includes atomic expressions
- At each precedence level use
  - Left recursion for left-associative operators
  - Right recursion for right-associative operators
  - No recursion for non-associative operators