CSE401: Parsing

Larry Snyder
Spring 2003

Slides by Chambers, Eggers, Naftin, Philcox, Snyder and others
© L. Snyder & UW CSE 1994-2003

Objectives: parsing lectures

Understand:
- Theory and practice of parsing
- Underlying language theory (CFGs, ...)
- Top-down parsing (and be able to do it)
- Bottom-up parsing (time permitting)
- Today’s focus: grammars and ambiguity

Parsing

- Abstract Syntax Tree (AST)
  - Captures hierarchical structure of the program
  - 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

- Is the program syntactically correct?
- If so, build the corresponding AST

Context-free grammars (CFGs)

- For lexing, we used regular expressions as the underlying notation
- For parsing, we use context-free grammars in much the same way
  - Regular expressions are not powerful enough
    - Intuitively, can’t express balance/nesting (if/then, parents)
    - More general grammars are more powerful than we need
  - We could use more power, but instead we delay some checking to semantic analysis instead of doing all the analysis based on the (general, but slow) grammar

CFG terminology

- Terminals: alphabet, or set of legal tokens
- Nonterminals: represent abstract syntax units
- Productions: rules defining nonterminals in terms of a finite sequence of terminals and nonterminals
- Start symbol: root symbol defining the language

Program ::= Stmt
Stmt ::= if Exp then Stmt else Stmt end
Stmt ::= while Exp do Stmt end
EBNF description of PL/0

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

EBNF description of PL/0

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

Exercise: produce a syntax tree for squares.

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

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/productions representing structure

Parse trees

- We represent derivations and parses as parse trees
- Concrete syntax tree
  - Exact reflection of the grammar
- Abstract syntax tree
  - Simplified version, reflecting key structural information
  - E.g., omit superfluous punctuation & keywords
Concrete Syntax Tree

Abstract syntax trees

Ex: An expression grammar

- $E ::= E \text{ Op } E | - E | \{E\} | \text{int}$
- $\text{Op ::= + | - | * | /}$

Using this grammar, find parse trees for:
- $3 \times 5$
- $3 + 4 \times 5$

Ambiguity

- Some grammars are ambiguous
  - Different parse trees with the same final string
  - (Some languages are ambiguous, with no possible non-ambiguous grammar, but we avoid them)
  - 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

- $\text{Stat ::= ... | if Expr then Stat | if Expr then Stat else Stat}$
- $\text{if el if el2 then sl else s2}$

- 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 ambiguity: #1

- Add a meta-rule
  - For instance, “else associates with the closest previous unmatched if”
- This works and keeps the original grammar intact
- But it’s ad hoc and informal
Resolving ambiguity: #2

- Rewrite the grammar to resolve it explicitly

\[
\text{Stmt} \; :\; :\; = \; \text{MatchedStmt} \; | \; \text{UnmatchedStmt}
\]

\[
\text{MatchedStmt} \; :\; :\; = \; \text{if Expr then MatchedStmt \; else MatchedStmt}
\]

\[
\text{UnmatchedStmt} \; :\; :\; = \; \text{if Expr then Stmt \; else UnmatchedStmt}
\]

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

Resolving ambiguity: #2 (cont.)

\[
\text{Stmt} \; :\; :\; = \; \text{MatchedStmt} \; | \; \text{UnmatchedStmt}
\]

\[
\text{MatchedStmt} \; :\; :\; = \; \text{if Expr then MatchedStmt \; else UnmatchedStmt}
\]

\[
\text{UnmatchedStmt} \; :\; :\; = \; \text{if Expr then Stmt \; else UnmatchedStmt}
\]

\[
\text{if } s_1 \text{ then if } s_2 \text{ then } s_1 \text{ else } s_2
\]

Resolving ambiguity: #3

- Redesign the programming language to remove the ambiguity

\[
\text{Stmt} \; :\; :\; = \; \text{if Expr then Stmt \; else Stmt}
\]

- 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 #3: redefine the language

Option #1: add meta-rules

- Add meta-rules for precedence and associativity

\[
E \; :\; :\; = \; E+E \mid E*E \mid E\cdot E \mid (E) \mid \ldots
\]

- *,\cdot < \cdot \cdot < \text{unary} < \cdot \quad \text{etc.}
- \cdot\cdot,\cdot left-associative; \cdot right associative
- Simple, intuitive
- But not all parsers can support this
  - yacc does

Option #2: new BNF

- 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
Option #2: example

\[ w + x + y \cdot z \]

\[ E ::= E + T \mid T \]
\[ T ::= T \cdot F \mid F \]
\[ F ::= \text{id} \mid (E) \]

Option #3: New language

- Require parens
  - E.g., in APL all exprs evaluated left-to-right unless parenthesized
- Forbid parens
  - E.g.: RPN calculators

Designing a grammar:

- on what basis?
- Accuracy
- Readability, clarity
- Unambiguity
- Limitations of CFGs
- Similarity to desired AST structure
- Ability to be parsed by a particular parsing algorithm
  - Top-down parser \(\rightarrow\) LL(k) grammar
  - Bottom-up parser \(\rightarrow\) LR(k) grammar

Parsing algorithms

- Given input (sequence of tokens) and grammar, how do we find an AST that represents the structure of the input with respect to that grammar?
- Two basic kinds of algorithms
  - Top-down: expand from grammar's start symbol until a legal program is produced
  - Bottom-up: create sub-trees that are merged into larger sub-trees, finally leading to the start symbol

Top-down parsing

- Build AST from top (start symbol) to leaves (terminals)
  - Represents a leftmost derivation (e.g., always expand leftmost non-terminal)
- Basic issue: when replacing a non-terminal with a right-hand side (rhs), which rhs should you use?
- Basic solution: Look at next input tokens

Predictive parser

- A top-down parser that can select the correct rhs looking at the next \(k\) tokens (lookahead)
- Efficient
  - No backtracking is needed
  - Linear time to parse
- Implementation
  - Table-driven: pushdown automaton (PDA) — like table-driven FSA plus stack for recursive FSA calls
  - Recursive-descent parser [used in PL/I]
    - Each non-terminal parsed by a procedure
    - Call other procedures to parse sub-non-terminals, recursively
LL(k), LR(k), …?

- These parsers have generally snazzy names
- The simpler ones look like the ones in the title of this slide
  - The first L means "process tokens left to right"
  - The second letter means "produce a (Right/Left)most derivation"
  - The k means "k tokens of lookahead"
- We won’t discuss LALR(k), SLR, and lots more parsing algorithms

LL(k) grammars

- It’s easy to construct a predictive parser if a grammar is LL(k)
  - Left-to-right scan on input
  - Leftmost derivation, k tokens of lookahead
- Restrictions include
  - Unambiguous
  - No common prefixes of length ≤ k
  - No left recursion
  - … (more details later)…
- Collectively, the restrictions guarantee that, given k input tokens, one can always select the correct rhs to expand

Eliminating common prefixes

- **Left factor** them, creating a new non-terminal for the common prefix and/or different suffixes
- **Before**
  - If : = if Test then States end |
  - if Test then States else States end
- **After**
  - If : = if Test then States ICnt |
  - ICnt := end |
- Grammar is a bit uglier
- Easy to do manually in a recursive-descent parser

Eliminating left recursion:

- **Before**
  - E := E + T | T
  - T := T * F | F
  - F := id | ( E )
- **After**
  - E := T ECnt
  - ECnt := + T ECnt |
  - T := F TCnt
  - TCnt := * F TCnt |
  - F := id |

Just add sugar

- Sugared form is still pretty readable
- Easy to implement in hand-written recursive descent parser
- Concrete syntax tree is not as close to abstract syntax tree

LL(1) Parsing Theory

**Goal:** Formal, rigorous description of those grammars for which "I can figure out how to do a top-down parse by looking ahead just one token", plus corresponding algorithms.

**Notation:**
- T = Set of Terminals (Tokens)
- N = Set of Nonterminals
- $S$ = End-of-file character (T-like, but not in N \cup T)
### Table-driven predictive parser

- Automatically compute PREDICT table from grammar
- PREDICT(nonterminal,input-symbol)
  1. action, e.g. which rhs or error

### Table - driven predictive parser

```plaintext
Automatically compute PREDICT table from grammar
PREDICT(nonterminal,input-symbol)
1. action, e.g. which rhs or error
```

### Example 1

```plaintext
Stmt ::= 1 if expr then Stmt else Stmt | 2 while Expr do Stmt | 3 begin Stmts end
Stmts ::= 4 Stmt ; Stmts | $ e
Expr ::= 6 id
```

<table>
<thead>
<tr>
<th>if then</th>
<th>else while</th>
<th>do begin end id</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stmt</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Stmts</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Expr</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Empty = error

### LL(1) Parsing Algorithm

- push S$
- push S$
  /* S is start symbol */
- while Stack not empty
  X := pop(Stack)
  a := peek at next token
  /* assume EOF = $ */
  if X is terminal or $
  If X=a, read token a else abort;
  else look at PREDICT(X, a)
  /* X is nonterminal */
  Empty : abort
  rule X → α : push α
  If not at end of input, abort

### LL(1) Parsing Algorithm

- push S$
  /* S is start symbol */
- while Stack not empty
  X := pop(Stack)
  a := peek at next token
  /* assume EOF = $ */
  if X is terminal or $
  If X=a, read token a else abort;
  else look at PREDICT(X, a)
  /* X is nonterminal */
  Empty : abort
  rule X → α : push α
  If not at end of input, abort

### Constructing PREDICT: overview

- Compute FIRST set for each rhs
  - All tokens that can appear first in a derivation from that rhs
  - In case rhs can be empty, compute FOLLOW set for each non-terminal
  - All tokens that can appear right after that non-terminal in a derivation
  - Constructions of FIRST and FOLLOW sets are interdependent
  - PREDICT depends on both

### Example 1 (cont.)

#### FIRST(α) – 1st “token” from α

Definition: For any string α of terminals and non-terminals, FIRST(α) is the set of terminals that begin strings derived from α, together with ε, if α can derive ε. More precisely:

For any α ∈ (N ∪ T)*,
FIRST(α) = \{ a ∈ T | α ⇒* a β for some β ∈ (N ∪ T)* \} ∪ {ε, if α ⇒* ε}
Computing FIRST – 4 cases

1. FIRST(c) = {c}
2. For all a ∈ T, FIRST(a) = {a}
3. For all A ∈ N, repeat until no change
   If there is a rule A → ε, add ε to FIRST(A)
   For all rules A → Y₁…Yₖ add FIRST(Y₁) - {ε}
   if ε ∈ FIRST(Y₁) then add FIRST(Y₂) - {ε}
   if ε ∈ FIRST(Y₁,Y₂) then add FIRST(Y₃) - {ε}
   ... if ε ∈ FIRST(Y₁,Y₂…Yₖ) then add(ε)

Computing FIRST (Cont.)

4. For all any string Y₁…Yₖ∈ (N ∪ T)*, similar:
   add FIRST(Y₁) - {ε}
   if ε ∈ FIRST(Y₁) then add FIRST(Y₂) - {ε}
   if ε ∈ FIRST(Y₁,Y₂) then add FIRST(Y₃) - {ε}
   ... if ε ∈ FIRST(Y₁,Y₂…Yₖ) then add(ε)

[Note: defined for all strings; really only care about FIRST(right hand sides.)]

FOLLOW(B) – Next “token” after B

Definition: for any non-terminal B, FOLLOW(B) is the set of terminals that can appear immediately after B in some derivation from the start symbol, together with $, if B can be the end of such a derivation. (ε represents “end of input”.)

More precisely: For all B ∈ N,

FOLLOW(B) = { a ∈ (T ∪ {$}) | S$ ⇒* a B for some α, β ∈ (N ∪ T ∪ {$})* } (S is the start symbol of the grammar.)

Computing FOLLOW(B)

Add $ to FOLLOW(S)
Repeat until no change
For all rules A → αB [i.e. all rules with a B in r.h.s],
Add FIRST(β) - {ε} to FOLLOW(B)
If ε ∈ FIRST(β) [in particular, if β is empty] then
Add FOLLOW(A) to FOLLOW(B)

Assume for all A that S ::= αA for some α ∈ (N ∪ T), else A irrelevant

Properties of LL(1) Grammars

- Clearly, given a conflict-free PREDICT table (≤ 1 entry/cell), the parser will do something unique with every input
- Key fact is, if the table is built as above, that something is the correct thing
- I.e., the PREDICT table will reliably guide the LL(1) parsing algorithm so that it will
  - Find a derivation for every string in the language
  - Declare an error on every string not in the language
Exercises (1st especially recommended)

- Easy:
  - Pick some grammar with common prefixes, left recursion, and/or ambiguity.
  - Build PREDICT; it will have conflicts
  - Harder: prove that every grammar with ≥1 of those properties will have PREDICT conflicts
  - Harder: Find a grammar with none of those features that nevertheless gives conflicts.
  - i.e., absence of those features is necessary but not sufficient for a grammar to be LL(1).
  - Harder, for theoryheads: if the table has conflicts, and the parser chooses among them nondeterministically, it will work correctly.

Example 2

\[
E ::= T \{ + T \} \\
T ::= F \{ * F \} \\
F ::= \text{id} | (E) \\
\]

Example 2 (cont.)

<table>
<thead>
<tr>
<th>PRED</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>T, T'</td>
</tr>
<tr>
<td>T'</td>
<td>+, T, T'</td>
</tr>
<tr>
<td>S</td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>F, T'</td>
</tr>
<tr>
<td>T'</td>
<td>* , F, T'</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>S</td>
<td>$</td>
</tr>
</tbody>
</table>

Example 2: PREDICT

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{id} & + & - & \ast & / & ( & ) & \$ \\
\hline
E    &   &   &   &   &   &   &   \\
E'   &   &   &   &   &   &   &   \\
F    &   &   &   &   &   &   &   \\
F'   &   &   &   &   &   &   &   \\
\hline
\end{array}
\]

PREDICT and LL(1)

- The PREDICT table has at most one entry in each cell if and only if the grammar is LL(1)
- There is only one choice (it’s predictive), making it fast to parse and easy to implement
- Multiple entries in a cell
  - Arise with left recursion, ambiguity, common prefixes, etc.
  - Can patch by hand, if you know what to do
  - Or use more powerful parser (LL(2), or LR(k), or...?)
  - Or change the grammar

Recursive descent parsers

- Write procedure for each non-terminal
- Each procedure selects the correct right-hand side by peeking at the input tokens
- Then the r.h.s. is consumed
  - If it’s a terminal symbol, verify it is next and then advance through the token stream
  - If it’s a non-terminal, call corresponding procedure
- Build and return AST representing the r.h.s.
Recursive descent example

```
Stat := if expr then Stmt else Stmt |
  while Expr do Stmt |
  begin Stmt end
Stms ::= Stmt ; Stmts |
Expr ::= id |
```

```
ParseStat():
  switch (next token) {
    "begin": ParseStmt(); read "end"; break;
    "while": ParseExpr(); read "do"; ParseStmt(); break;
    "if": ParseExpr(); read "then"; ParseStmt();
      read "else"; ParseStmt(); break;
    default: abort;
  }
```

```
Parser::ParseStmts() {
  StmtArray* Parser::ParseStmts() {
    StmtArray* states = new StmtArray; Stmt* stmt;
    for (;;) {
      Token t = scanner->Peek();
      switch (t->kind) {
        case IDENT: stmt = ParseIdentStmt(); break;
        case OUTPUT: stmt = ParseOutputStmt(); break;
        case IF: stmt = ParseIfStmt(); break;
        case WHILE: stmt = ParseWhileStmt(); break;
        default: return stmt; // no more states
      }
      states->add(stmt);
      scanner->Read(SEMICOLON);
    }
  }
```

LL(1) and Recursive Descent

- If the grammar is LL(1), it's easy to build a recursive descent parser
  - One nonterminal/row one procedure
  - Use 1 token lookahead to decide which rhs
  - Table-driven parser's stack recursive call stack
- Recursive descent can handle some non-LL(1) features, too.

Example LL(1) & recursive descent

```
Stat := if expr then Stmt else Stmt |
  while Expr do Stmt |
  begin Stmt end
Stms ::= Stmt ; Stmts |
Expr ::= id |
```

Example non-LL(1) & recursive descent

```
Stat := if expr then Stmt |
  while Expr do Stmt |
  begin Stmt end
Stms ::= Stmt ; Stmts |
Expr ::= id |
```

It's demo time…

Let's look at some of the PL/O code to see how the recursive descent parsing works in practice
Parser::ParseIfStmt()

Stmt* Parser::ParseIfStmt() {
    scanner->Read(IF);
    Exp* test = ParseTest();
    scanner->Read(THEN);
    StmtArray* stmts = ParseStmts();
    scanner->Read(END);
    return new IfStmt(test, stmts);
}

Parser::ParseWhileStmt()

Stmt* Parser::ParseWhileStmt() {
    scanner->Read(WHILE);
    Exp* test = ParseTest();
    scanner->Read(DO);
    StmtArray* stmts = ParseStmts();
    scanner->Read(END);
    return new WhileStmt(test, stmts);
}

Parser::ParseIdentStmt()

Stmt* Parser::ParseIdentStmt() {
    Token* id = scanner->Read(IDENT);
    if (scanner->CondRead([LPAREN]) {
        Exp* args = ParseExpr();
        scanner->Read(RPAREN);
        return new CallStmt(id->ident(), args);
    } else {
        LValue* lv = new VarRef(id->ident());
        scanner->Read(GET);
        return new AssignStmt(lv, ParseExpr());
    }
}

Yacc — A bottom-up-parser generator

"yet another compiler-compiler"

Input:
- grammar, possibly augmented with action code

Output:
- C code to parse it and perform actions
- LALR(1) parser generator
- practical bottom-up parser
- more powerful than LL(1)
- modern updates of yacc
- yacc++, bison, byacc, …
Yacc input grammar

```plaintext
assignstat: IDENTITY GETS expr
ifstmt: IF test THEN stmts END
| IF test THEN stmts ELSE stmts END
expr: term
| expr '+' term
| expr '-' term
factor: '-' factor
| IDENTITY
| INTEGER
| INPUT
| '(' expr ')' 
```

Yacc with actions

```plaintext
assignstat: IDENTITY GETS expr { $0 = new AssignStmt($1, $3); }
ifstmt: IF be THEN stmts END { $0 = new IfStmt($2, $4, $6); }
| IF be THEN stmts ELSE stmts END { $0 = new IfStmt($2, $4, $6); }
expr: term { $0 = $1; }
| expr '+' term { $0 = new BinOp(PLUS, $1, $3); }
| expr '-' term { $0 = new BinOp(MINUS, $1, $3); }
factor: '-' factor { $0 = new UnOp(MINUS, $2); }
| IDENTITY { $0 = new VarDef($1); }
| INTEGER { $0 = new IntLiteral($1); }
| INPUT { $0 = new InputExpr; }
| '(' expr ')' { $0 = $2; }
```

Parsing summary

- Discover/impose a useful (hierarchical) structure on flat token sequence
- Represented by Abstract Syntax Tree
- Validity check syntax of input
- Could build concrete syntax tree (but don't)
- Many methods available
  - Top-down: LL(1)/recursive descent common for simple, by-hand projects
  - Bottom-up: LR(1)/LALR(1)/SLR(1) common for more complex projects
  - parser generator (e.g., yacc) almost necessary

Objectives: today

- Ambiguity
- Issues in designing a grammar
- AST extensions for the 401 project
- Overview of parsing algorithms
- Motivation and details of top-down, predictive parsers
- Recursive descent parsing
- Today++: a walk through the PL/0 parser
EBNF description of PL/0

Program : = module Id ; Block Id .
Block : = DeclList begin StmtList end
DeclList : = { Decl ;...}
Decl : = ConstDecl | ProcDecl | VarDecl
ConstDecl : = const DeclItem { , ConstDeclItem }
DeclItem : = Id : Type = ConstExpr
ConstExpr : = Id | Integer
VarDecl : = var DeclItem { , DeclItem }
Decl : = Id : Type
ProcDecl : = procedure Id ( [ FormalDecl { , FormalDecl } ] ) ; Block Id

FormalDecl : = Id : Type
StmtList : = { Stmt ;...}
Stmt : = CallStmt | AssignStmt | OutStmt | IfStmt | WhileStmt
CallStmt : = Id ( [ Exprs ] )
AssignStmt : = Lvalue : = Expr
Lvalue : = Id
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 )

AST extensions in project

Expressions
- true and false constants
- array index expression (an index)
- function call expression
- and or operators
- tests are expressions
- constant expressions

Statements
- for
- break
- return
- if with else

Declarations
- procedures with result types
- var parameters

Types
- Bool
- array