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?
  a := 3 * (5 + 4);  if x > y then m := x;
  a := 3 * / 4;     if x < y else m := x;
- 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 (a/b/c, parens)
    - 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 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 ::= DeclList | ProcDecl | VarDecl
ProcDecl ::= const ConstDeclItem ; ConstDecl
ConstDeclItem ::= Id : Type = ConstExpr
ConstExpr ::= Id | Integer
VarDecl ::= var VarDeclItem , VarDeclItem
VarDeclItem ::= Id : Type

EBNF description of PL/0

ProcDecl ::= procedure Id ( FormalDecl ) ;
FormalDecl ::= Id : Type
Type ::= int
StmtList ::= [ Stmt ; ]
Stmt ::= CallStmt | AssignStmt | OutStmt |
IfStmt | WhileStmt
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 )

Exercise: produce a syntax tree for squares.0

derive the syntax tree for:

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/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

Ambiguity

Another famous ambiguity: dangling else

Resolving ambiguity: #1
Resolving ambiguity: #2

- Rewrite the grammar to resolve it explicitly

```plaintext
Stmt ::= MatchedStmt | UnmatchedStmt
MatchedStmt ::= ...
  if Expr then MatchedStmt
  else MatchedStmt
UnmatchedStmt ::= if Expr then Stmt |
  if Expr then MatchedStmt
  else UnmatchedStmt
```

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

Resolving ambiguity: #2 (cont.)

```plaintext
if e1 then if e2 then s1 else s2
```

Resolving ambiguity: #3

- Redesign the programming language to remove the ambiguity

```plaintext
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 #3: redefine the language

Option #1: add meta-rules

- Add meta-rules for precedence and associativity

```plaintext
E ::= E+E | E-E | E*E | E/E | E^E | (E) | -E | ...
```

- +,- < */ < unary - < ^ etc.
- +,-/ left-associative; ^ 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

```plaintext
E ::= E+T | T
T ::= T*F | F
F ::= id | (E)
```
Option #2: example

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

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/0]
    - 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"
    - Leftmost => top-down
    - Rightmost => bottom-up
  - The k means "k tokens of lookahead"
- We won’t discuss LALR(k), SLR, and lots more parsing algorithms

Eliminating common prefixes

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

Just add sugar

- E ::= T ( + T )
- T ::= F ( * F ) |
- F ::= id | ( E ) |

- 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(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 left recursion:

- Before
  - E ::= E + T | T
  - T ::= T * F | F
  - F ::= id | ( E ) |
- After
  - E ::= T ECont
  - ECont ::= + T ECont |
  - T ::= F TCont
  - TCont ::= * F TCont |
  - F ::= id | ( E ) |

- Left recursion
  - E ::= E op E |

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
- $ = End-of-file character (T-like, but not in N∪T)
Table-driven predictive parser

- Automatically compute PREDICT table from grammar
- \text{PREDICT}(\text{nonterminal}, \text{input-symbol}) \rightarrow \text{action}, \text{e.g. which rhs or error}

Example 1

\text{Stmt} ::= 1 \text{if expr then Stmt else Stmt} | 2 \text{while Expr do Stmt} | 3 \text{begin Stmts end}

\text{Stmts} ::= 4 \text{Stmt ; Stmts} | 5 \varepsilon

\text{Expr} ::= 6 \text{id}

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.)

\begin{array}{|c|c|c|c|c|c|c|}
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\text{if} & \text{else} & \text{while} & \text{do} & \text{begin} & \text{end} & \text{id} \\
\hline
\text{Stmt} & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\text{Stmts} & 4 & 4 & 4 & 5 & 5 & 5 \\
\hline
\text{Expr} & 6 & 6 & 6 & 6 & 6 & 6 \\
\hline
\text{empty} = \text{error} \\
\hline
\end{array}

FIRST(\alpha) – 1st “token” from \alpha

Definition: For any string \alpha of terminals and non-terminals, \text{FIRST}(\alpha) is the set of terminals that begin strings derived from \alpha, together with \varepsilon, if \alpha can derive \varepsilon. More precisely:

For any \alpha \in (N \cup T)^*,
\text{FIRST(\alpha)} = 
\{ a \in T \mid \alpha \Rightarrow^* a \beta \text{ for some } \beta \in (N \cup T)^* \} \cup 
\{ \varepsilon, \text{if } \alpha \Rightarrow^* \varepsilon \}
Computing FIRST – 4 cases

1. \( \text{FIRST}(\epsilon) = \{ \epsilon \} \)
2. For all \( a \in T \), \( \text{FIRST}(a) = \{ a \} \)
3. For all \( A \in N \), repeat until no change
   - If there is a rule \( A \rightarrow \epsilon \), add(\( \epsilon \)) to \( \text{FIRST}(A) \)
   - For all rules \( A \rightarrow Y_1 \ldots Y_k \), add(\( \text{FIRST}(Y_1) - \{ \epsilon \} \)) if \( \epsilon \in \text{FIRST}(Y_1) Y_2 \ldots Y_k \)

Computing FIRST (Cont.)

4. For all any string \( Y_1 \ldots Y_k \in (N \cup T)^* \), similar:
   - add(\( \text{FIRST}(Y_1) - \{ \epsilon \} \)) if \( \epsilon \in \text{FIRST}(Y_1) \)
   - if \( \epsilon \in \text{FIRST}(Y_1 Y_2) \) then add(\( \text{FIRST}(Y_3) - \{ \epsilon \} \))
   - if \( \epsilon \in \text{FIRST}(Y_1 Y_2 \ldots Y_k) \) then add(\( \epsilon \))

[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 the "$", if \( B \) can be the end of such a derivation. (\$ represents “end of input”.) More precisely: For all \( B \in N \),

\[
\text{FOLLOW}(B) = \{ a \in (T \cup \{\$\}) \mid S \Rightarrow^* \alpha B a \beta \text{ for some } \alpha, \beta \in (N \cup T \cup \{\$\})^* \}
\]

(S is the Start symbol of the grammar.)

Computing FOLLOW(B)

Add $ to FOLLOW(S)
Repeat until no change
For all rules \( A \rightarrow \alpha B \beta \) [i.e. all rules with a B in r.h.s],
Add (\( \text{FIRST}(B) - \{ \epsilon \} \)) to FOLLOW(B)
If \( \epsilon \in \text{FIRST}(B) \) [in particular, if \( B \) is empty]
   then Add FOLLOW(A) to FOLLOW(B)
Assume for all \( A \) that \( S \Rightarrow^* \alpha A \beta \) for some \( \alpha, \beta \in (N \cup T)^* \), else \( A \) irrelevant

Properties of LL(1) Grammars

- Clearly, given a conflict-free PREDICT table (\( \leq 1 \) 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

Defn: \( G \) is LL(1) iff every cell has \( \leq 1 \) entry
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 ::= − F | id | ( E )

Example 2 (cont.)

<table>
<thead>
<tr>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>E ::= T E’</td>
<td></td>
</tr>
<tr>
<td>E’ ::= + T E’</td>
<td></td>
</tr>
<tr>
<td>T ::= F T’</td>
<td>id</td>
</tr>
<tr>
<td>T’ ::= * F T’</td>
<td>( E )</td>
</tr>
<tr>
<td>F ::= − F</td>
<td></td>
</tr>
</tbody>
</table>

Example 2: PREDICT

<table>
<thead>
<tr>
<th>id</th>
<th>+</th>
<th>−</th>
<th>*</th>
<th>/</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PREDICT and LL(1)

- The PREDICT table has at most one entry in each cell if and only if the grammar is LL(1)
  - i.e., 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

```
Stmt ::= if expr then Stmt else Stmt |
       while Expr do Stmt |
       beginStmts end
Stmts ::= Stmt ; Stmts |
       ε
Expr ::= id

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

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

```
Stmt ::= if expr then Stmt else Stmt |
       while Expr do Stmt |
       beginStmts end
Stmts ::= Stmt ; Stmts |
       ε
Expr ::= id

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

Example

```
Stmt ::= if expr then Stmt else Stmt |
       while Expr do Stmt |
       beginStmts end
Stmts ::= Stmt ; Stmts |
       ε
Expr ::= id

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

It's demo time…

- Let's look at some of the PL/0 code to see how the recursive descent parsing works in practice
Parser::ParseIfStmt()

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

<if stmt> ::= if <test> then <stmt list> end

Parser::ParseWhileStmt()

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

<while stmt> ::= while <test> do <stmt list> end

Parser::ParseIdentStmt()

Stmt* Parser::ParseIdentStmt() {
    Token* id = scanner->Read(IDENT);
    if (scanner->CondRead(LPAREN)) {
        ExprArray* args;
        if (scanner->CondRead(RPAREN)) {
            args = NULL;
        } else {
            args = ParseExprs();
            scanner->Read(RPAREN);
        }
        return new CallStmt(id->ident(), args);
    } else {
        LValue* lvalue = new VarRef(id->ident());
        scanner->Read(GETS);
        return new AssignStmt(lvalue, ParseExpr());
    }
}

{id stmt} ::= <call stmt> | <assign stmt>
<call stmt> ::= IDENT "(" [ <exprs> ] ")"
<assign stmt> ::= <lvalue> := <expr>
<lvalue> ::= IDENT

Parser::ParseSum()

Expr* Parser::ParseSum() {
    Expr* expr = ParseTerm();
    for (;;) {
        Token* t = scanner->Peek();
        if (t->kind() == PLUS || t->kind() == MINUS) {
            scanner->Get(); // eat the token
            Expr* expr2 = ParseTerm();
            expr = new BinOp(t->kind(), expr, expr2);
        } else {
            return expr;
        }
    }
}

<sum> ::= <term> { (+ | -) <term> }

Parser::ParseTerm()

Expr* Parser::ParseTerm() {
    Expr* expr = ParseFactor();
    for (;;) {
        Token* t = scanner->Peek();
        if (t->kind() == PLUS || t->kind() == MUL || t->kind() == DIVIDE) {
            scanner->Get(); // eat the token
            Expr* expr2 = ParseFactor();
            expr = new BinOp(t->kind(), expr, expr2);
        } else {
            return expr;
        }
    }
}

<term> ::= <factor> { (* | /) <factor> }

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

assignstmt: IDENT GETS expr
ifstmt: IF test THEN stmts END | IF test THEN stmts ELSE stmts END
expr: term | expr '+' term | expr '-' term
factor: '-' factor | IDENT | INTEGER | INPUT | '(' expr ')' 

Yacc with actions

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

Building a table-driven predictive parser

- Context-free grammars
  - Definitions
  - Manipulations (algorithmic)
    - Left factor common prefixes
    - Eliminating left recursion
  - Ambiguity & (semi-heuristic) fixes
    - meta-rules (code/precedence tables)
    - rewrite grammar
    - change language