Syntax Analysis/Parsing

Purpose:
- determine if tokens have the right form for the language (right syntactic structure)
- stream of tokens ⇒ abstract syntax tree (AST)

AST:
- captures hierarchical structure of input program
- a primary representation of program

Context-free grammars (CFG’s)

Syntax specified using CFG’s
- capture important structural characteristics

Notation for CFG’s: Backus Normal (Naur) Form (BNF)
- set of terminal symbols (tokens from lexical analysis)
- set of nonterminals (sequences of terminals &/or nonterminals)
- impose the hierarchical structure
- set of productions combine terminals & nonterminals
- nonterminal ::= nonterminals &/or terminals
- start symbol: nonterminal that denotes the language

CFG: set of productions that define a language

BNF description of PL/0 syntax

Program ::= module Id ;
Block ::= begin StmtList end;
StmtList ::= { Stmt; }
Stmt ::= CallStmt | AssignStmt | OutStmt | IfStmt | WhileStmt
CallStmt ::= Id ( Exprs )
AssignStmt ::= LValue := Expr
LValue ::= Id
OutStmt ::= output := Expr
IfStmt ::= if Test then StmtList else 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 )

Context-free grammars vs. Regular Expressions

CFG can check everything a RE can but:
- not need CFG power for lexical analysis
- REs are a more concise notation for tokens
- lexical analyzers constructed automatically are more efficient
- more modular front end

RE’s not powerful enough for parsing
- nested constructs
- recursion

Derivations & Parse Trees

Example grammar

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

(a + b * -c) * d

Derivation:
- define the language specified by the grammar
- sequence of expansion steps, beginning with start symbol, leading to a string of terminals
- production seen as rewriting rule: nonterminal replaced by the rhs

Parsing: inverse of derivation
- given target string of terminals (tokens), want to recover nonterminals representing structure

Can represent derivation as a:
- parse tree (concrete syntax tree)
- graphical representation for a derivation
- keeps the grammar symbols
- don’t record the expansion order
- abstract syntax tree (AST)
- simpler representation
- precedence implied by the hierarchy
Abstract syntax trees represent only important aspects of concrete syntax trees
- no need for “sign posts” like (,),;,,do,end
- rest of compiler only cares about abstract structure
- can regenerate concrete syntax tree from AST when needed

### AST extensions in project

**Expressions:**
- true and false constants
- array index expression
- function call expression
- and, or operators
- tests are expressions
- constant expressions

**Statements:**
- for statement
- return stmt
- if stmt with else
- array assignment stmt (similar to array index expression)

**Declarations:**
- procedures with result type
- var parameters (passed by reference)

**Types:**
- boolean type
- array type

---

### Parsing algorithms

**Given grammar, want to see if an input program can be generated by it**
- check legality
- produce AST representing structure
- be efficient

**Kinds of parsing algorithms:**
- top-down ⇒ LL(k) grammar
- bottom-up ⇒ LR(k) grammar
  - Left to right scan on input
  - Leftmost/Rightmost derivation can see k tokens at once

---

### Top-down parsing

Build parse tree for input program from the top (start symbol) down to leaves (terminals)
- find leftmost derivation for an input string (replace the leftmost nonterminal at each step)
- create parse tree nodes in preorder

**Basic issue:**
- when replacing a nonterminal with some rhs, how to pick which rhs?

E.g.

- **Stmt**: Call | Assign | If | While
- **Call**: Id
- **Assign**: Id := Expr
- **If**: if Test then Stmts end
- **While**: while Test do Stmts end

**Solution:** look at input tokens to help decide

---

### Predictive parsing

**Predictive parser:**
- top-down parser that can select correct rhs looking at at most k input tokens (the look-ahead)

**Efficient:**
- no backtracking needed
- linear time to parse

**Implementation of predictive parsers:**
- **table-driven parser**
  - like table-driven FSA
  - plus stack to hold productions, recursively
- **recursive descent parser**
  - each nonterminal parsed by a procedure
  - call other procedures to parse sub-nonterminals, recursively
LL(k) grammars

Can construct predictive parser automatically/easily if grammar is LL(k)

- Left-to-right scan of input. Leftmost derivation
- k tokens of look-ahead needed (k ≥ 1) to make parsing decisions

Some restrictions:
- no ambiguity
  >1 parse tree (derivation) for a sentence in the language
- no common prefixes of length ≥ k
  S := if Test then Ss end | ...
  if Test then Ss else Ss end | ...
- no left recursion
  E := E Op E | ...
  a few others

Restrictions guarantee that, given k input tokens, can always select correct rhs to expand nonterminal

Ambiguity

Some grammars are ambiguous:
- multiple parse trees with same final string
- produces more than one leftmost or rightmost derivation

Structure of parse tree captures much of meaning of program; ambiguity ⇒ multiple possible meanings for same program

Solutions:
1) add meta-rules
2) change the grammar
3) change the language

Famous ambiguities: “dangling else”

Stmt ::= ...
  if Expr then Stmt | ...
  if Expr then Stmt else Stmt

'if v1 then if v2 then s1 else s2'

Resolving the ambiguity

Option 1: add a meta-rule
- e.g. “else associates with closest previous then”
+ works
+ keeps original grammar intact
- ad hoc and informal

Option 2: rewrite the grammar to resolve ambiguity explicitly

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

+ formal, no additional rules beyond syntax
- sometimes obscures original grammar

Option 3: redesign the language to remove the ambiguity

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

+ extra end required for every if
+ formal, clear, elegant
+ changing the language
Another famous ambiguity: expressions

E ::= E Op E | E Op (E) | id
Op ::= * | / |

"a + b * c"

Resolving the ambiguity

Option 1: add some meta-rules, e.g. precedence and associativity rules

Example:

E ::= E Op E | E Op (E) | id
Op ::= * | / |

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -</td>
<td>highest</td>
<td>right</td>
</tr>
<tr>
<td>* /</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td>&lt; &gt; and</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>= and</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>lowest left</td>
<td></td>
</tr>
</tbody>
</table>

Resolving the ambiguity

Option 2: modify the grammar to explicitly resolve the ambiguity

Strategy:
- create a nonterminal for each precedence level
- expr is lowest precedence nonterminal
- each nonterminal can be rewritten with a higher precedence operator
- highest precedence operator includes terminals
- at each precedence level, use:
  - left recursion for left-associative operators
  - right recursion for right-associative operators
  - no recursion for non-associative operators

Example

Expr ::= Expr0
Expr0 ::= Expr0 or Expr1 | Expr1
Expr1 ::= Expr1 and Expr2 | Expr2
Expr2 ::= Expr3 (Expr3) Expr3 | Expr3
Expr3 ::= Expr4 (+) Expr4 | Expr4
Expr4 ::= Expr5 (Expr5) Expr5 | Expr5
Expr5 ::= Expr6 | Expr6
Expr6 ::= id | int | ... | (Expr0)

Eliminating common prefixes

Can left factor common prefixes to eliminate them
- create new nonterminal for common prefix and/or different suffixes

Before:

If ::= if Test then Stmt1 and |

After:

If ::= if Test then Stmt1 IfCont
IfCont ::= and | else Stmt1 and

Grammar a bit uglier
Easy to do by hand in recursive-descent parser

Eliminating left recursion

Can rewrite grammar to eliminate left recursion

Before:

E ::= T |
T ::= T * F |
F ::= id |

After:

E ::= T ECont
ECont ::= * T ECont | €
T ::= F TCont
TCont ::= * F TCont | €
F ::= id |

right-recursive productions
Transition Diagrams

“Railroad diagrams”
• another more graphical notation for CFGs
• diagram per nonterminal
• look like FSAs, where arcs can be labelled with nonterminals as well as terminals
  • if terminal: follow the arc
    parser gets a new token & compares it with the terminal on the arc
  • if nonterminal: go to new diagram
    parser calls the procedure for the nonterminal (recursive descent parser)

Table-driven predictive parser

Can automatically convert grammar into parsing table

PREDICT (nonterminal, input-sym) ⇒ production
• selects the right production to take given a nonterminal to expand and the next token of the input

Example:
 stmt ::= if expr then stmt else stmt |
        begin stmts end
 stmts ::= stmt ; stmts |
 expr ::= id

Parsing table

<table>
<thead>
<tr>
<th></th>
<th>if</th>
<th>then</th>
<th>else</th>
<th>while</th>
<th>begin</th>
<th>end</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stmts</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FOLLOW

Definition: For all nonterminals B in N, FOLLOW(B) is the set of terminals (or $) that can follow B in a derivation. I.e.,
FOLLOW(B) = \{ a \in (T \cup \{\$\}) \mid S \rightarrow^* a B \beta \text{ for some } x, \beta \in (N \cup T \cup \{\$\})^* \}

Computing FOLLOW

+ Add $ to FOLLOW(S)
+ Repeat until no change:
  For all rules A → α B β
  (i) add (FIRST(β) - {ε}) to FOLLOW(B)
  (ii) if ε in FIRST(β) \ [e.g. if β = ε]
       add FOLLOW(A) to FOLLOW(B)

Constructing PREDICT table

Start ::= S $
S ::= if \mid id
IF ::= if E then S IFCont
IFCont ::= else S \mid ε

FIRST FOLLOW

<table>
<thead>
<tr>
<th></th>
<th>if</th>
<th>else</th>
<th>while</th>
<th>do</th>
<th>begin</th>
<th>end</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stmts</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another example

S ::= E $
E ::= T E'
E' ::= (+ | -) T E' | ε
T ::= F T'
T' ::= (*/ | /) F T' | ε
F ::= = F | id | ( E )

FIRST (RHS) FOLLOW (X)

S ::= E $
E  ::= T E'
E' ::= (+ | -) T E' | ε
T ::= F T'
T' ::= (*/ | /) F T' | ε
F ::=

Recursive descent parsers

Write subroutine for each non-terminal
  • each subroutine first selects correct r.h.s. by peeking at
    input tokens
  • then consume r.h.s.
    • if terminal symbol, verify that it's next & then advance
    • if nonterminal, call corresponding subroutine
  • construct & return AST representing r.h.s.

Example

If ::= if Expr then Stmt [else Stmt] end ;

Stmt ::= IdStmt ;
IdStmt ::= CallStmt | AssignStmt;
CallStmt ::= IDENT "(" Expr ")";
AssignStmt ::= IDENT := Expr;

Example

ParseExp => ParseSum => ParseTerm => ParseFactor

Sum ::= Term { (+ | -) Term }
Term ::= Factor { (*/ | /) Factor }
Expr* Parser::ParseSum() {
  Expr* expr = ParseTerm();
  if (t-kind() == PLUS || t-kind() == MINUS)
    scanner->Get();...
  else { break; }
}

ParseFactor();
  for (ii) {
    Token* t = scanner->Peek();
    if (t-kind() == MUL || t-kind() == DIVIDE)
      scanner->Get();...
  else ( break;)
}

Expr* Parser::ParseFactor() {
  Expr* expr = ParseFactor();
  if (ii) {
    Token* t = scanner->Peek();
    if (t-kind() == MUL || t-kind() == DIVIDE)
      scanner->Get();...
  else ( break;)
  }
}

Example

PREDICT and LL(1)

If PREDICT table has at most one entry in each cell,
then grammar is LL(1)
  • always exactly one right choice
    fast to parse and easy to implement
  • LL(1) => each column labelled by 1 token
  
  Can have multiple entries in each cell
  • common prefixes
  • left recursion
  • ambiguity
Yacc

`yacc`: “yet another compiler” compiler

Input:
- grammar
- possibly augmented with action code

Output:
- C functions to parse grammar and perform actions

LALR parser generator
- practical bottom-up parser
- more powerful than LL(1)
- used for parser generators

`yacc++`, `bison`, `byacc` are modern updates of `yacc`

Yacc input grammar

Example declaration:
```
%{
#include <stdio.h>
%

%token INTEGER
```

Example grammar productions:
```
assignstmt: IDENT GETS expr { $$ = new AssignStmt($1, $3); };

ifstmt: IF test THEN stmts END { $$ = new IfStmt($2, $4); }
  | IF test THEN stmts ELSE stmts END { $$ = new IfElseStmt($2, $4, $6); };

expr: term { $$ = $1; }
  | expr '+' term { $$ = 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; }
```

Yacc with semantic actions

Example grammar productions:
```
assignstmt: IDENT GETS expr ( $$ = new AssignStmt($1, $3); )
  ;

ifstmt: IF test THEN stmts END ( $$ = new IfStmt($2, $4); )
  | IF test THEN stmts ELSE stmts END ( $$ = new IfElseStmt($2, $4, $6); )
  ;

expr: term ( $$ = $1; )
  | expr '+' term ( $$ = 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; )
```

Error handling

How to handle syntax error: error recovery

Option 1: quit compilation ⇒ PL/0
   + easy
   - inconvenient for programmer

Option 2: do more before quit
   + try to catch as many errors as possible on one compile
   - avoid streams of spurious errors

Option 3: error correction
   + fix syntax errors as part of compilation
   - hard!