**Bottom-up parsing**

Construct parse tree for input from leaves up
- **reducing** a string of tokens to single start symbol
  (inverse of deriving a string of tokens from start symbol)

"Shift-reduce" strategy:
- read ("shift") tokens until seen r.h.s. of "correct" production
- reduce r.h.s. to l.h.s. nonterminal, then continue
- done when all input read and reduced to start nonterminal

**LR(k) parsing**

LR(k) parsing algorithms
- Left-to-right scan of input, Rightmost derivation
- k tokens of lookahead

The most general kind of bottom-up parsing

Strictly more general than LL(k)
- gets to look at whole rhs of production before deciding what to do, not just first k tokens of rhs
- can handle left recursion and common prefixes fine

Still as efficient as any top-down or bottom-up parsing method

Complex to implement
- need automatic tools to construct parser from grammar

**LR parsing tables**

Construct parsing tables implementing a FSA plus a stack
- rows: states of parser
- columns: token(s) of lookahead
- entries: action of parser
  - shift, then goto state S
  - reduce production “LHS ::= RHS”
  - accept
  - error

Algorithm to construct FSA similar to algorithm to build DFA from NFA
- each state represents set of possible "places" in parsing

LR(k) algorithm builds big tables
LALR(k) algorithm has fewer states ⇒ smaller tables
- less general than LR(k), but still good in practice

Most parser generators, including yacc and cup, are LALR(1)

**LR(0) parser generation**

Key idea:
- simulate where input might be in grammar as it reads tokens

"Where input might be in grammar" captured by set of items,
which forms a state in the parser’s FSA
- LR(0) item: lhs ::= rhs production, with dot in rhs
  somewhere marking what’s been read (shifted) so far
- LR(k) item: also add k tokens of lookahead to each item

Example grammar:

```
S ::= beep | { L }
L ::= S | L ; S
```

Add an initial start production P ::= S $
- $ represents end-of-input

Initial item:

```
P ::= . S $
```
**Closure**

Initial state is **closure** of initial item
- closure: if dot before non-terminal, add all productions for non-terminal with dot at the start
- "epsilon transitions"

State 1:
```
P ::= . S $
S ::= . beep
S ::= . { L }
```

**State transitions**

Given set of items, compute new state(s) for each symbol (terminal and non-terminal) after dot
- state transitions correspond to shift actions

New item derived from old item by shifting dot over symbol
- then do closure of this item to compute new state

State 1:
```
P ::= . S $
S ::= . beep
S ::= . { L }
```

State 2 reached on transition that shifts S:
```
P ::= S . $
```

State 3 reached on transition that shifts beep:
```
S ::= beep .
```

State 4 reached on transition that shifts {:
```
S ::= . L )
L ::= . S
L ::= . L ; S
S ::= . beep
S ::= . { L }
```

**Reducing states**

If state has \( lhs ::= rhs . item \),
then the state has a **reduce** \( lhs ::= rhs \) action

Example:
```
State 3:
S ::= beep .
\text{reduce} \ S ::= beep
```

Conflicting actions?
- what if other items in this state shift?
- what if other items in this state reduce differently?

**Accepting states**

Special case:
- **reduce** \( P ::= ... \ $ . \) action replaced with **accept** action

Example:
```
State 2:
P ::= S . $
on $, shift and goto State 5
```

State 5:
```
P ::= S $ .
\text{accept}
```
Rest of the states (part 1)

State 4:
\[
S ::= \{ . \ L \}  \\
L ::= . \ S  \\
L ::= L ; S  \\
S ::= \text{beep}  \\
S ::= . \{ L \}  \\
\text{on beep, shift and goto State 3}  \\
\text{on }, shift and goto State 4  \\
\text{on S, shift and goto State 6}  \\
\text{on L, shift and goto State 7}
\]

State 6:
\[
L ::= S .  \\
\text{reduce} L ::= S
\]

State 7:
\[
S ::= \{ L . \}  \\
L ::= L . ; S  \\
\text{on }, shift and goto State 8  \\
\text{on ; , shift and goto State 9}
\]

Rest of the states (part 2)

State 8:
\[
S ::= \{ L \} .  \\
\text{reduce} S ::= \{ L \}
\]

State 9:
\[
L ::= L ; . S  \\
S ::= \text{beep}  \\
S ::= . \{ L \}  \\
\text{on beep, shift and goto State 3}  \\
\text{on }, shift and goto State 4  \\
\text{on S, shift and goto State 10}
\]

State 10:
\[
L ::= L ; S .  \\
\text{reduce} L ::= L ; S
\]

(whew)

Building LR(0) tables from the states & transitions

Represent state machine using two tables:
- action table and goto table
- each has a row per state

Action table: single column giving each state's action
(shift, reduce, or accept)

Goto table: one column for each terminal & non-terminal symbol

For every "state i on x, shift and goto state j" transition:
- put shift in row i of action table
- put goto j in row i, column x, of goto table

For every "state i, reduce lhs ::= rhs" action:
- put reduce lhs ::= rhs in row i of action table

For every "state i, accept" action:
- put accept in row i of action table

Better not put more than one action in any row!

Table for this grammar

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s</td>
<td>g4 g3 g2 g5</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r</td>
<td>S ::= beep</td>
</tr>
<tr>
<td>4</td>
<td>s</td>
<td>g4 g3 g6 g7</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r</td>
<td>L ::= S</td>
</tr>
<tr>
<td>7</td>
<td>s</td>
<td>g8 g9</td>
</tr>
<tr>
<td>8</td>
<td>r</td>
<td>S ::= { L }</td>
</tr>
<tr>
<td>9</td>
<td>s</td>
<td>g4 g3 g10</td>
</tr>
<tr>
<td>10</td>
<td>r</td>
<td>L ::= L ; S</td>
</tr>
</tbody>
</table>
Execution of parsing table

Parser state:
- stack of states, initialized to "1"
- \(<\text{shifted/reduced symbols}, <\text{unconsumed tokens}>\>,
  initialized to ", <\text{input tokens}>"

To run the parser, repeat these two steps:
- do \(\text{action}(S)\), where \(S\) is state on top of stack
- push \(\text{goto}(S,X)\) onto stack, where \(S\) is state on top of stack
  and \(X\) is symbol to left of ,
  if \(\text{goto}(S,X)\) empty, report syntax error

Semantics of actions:
shift:
- move first unconsumed token across , to end of shifted tokens
reduce \(LHS ::= RHS\)
- replace \([RHS]\) symbols from end of shifted/reduced symbols
  with \(LHS\)
- build parse tree node for \(LHS\) from \(RHS\) subtrees
- pop \([RHS]\) states from state stack
accept:
- done parsing! return parse tree

Example

\[
\{ \text{beep} ; ( \text{beep} ) \} \$

Problems in shift-reduce parsing

Can write grammars that cannot be handled with shift-reduce parsing
- ambiguous grammars will always have these problems
- some unambiguous grammars do, too

Shift/reduce conflict:
- state has both shift action(s) and reduce actions

Reduce/reduce conflict:
- state has more than one reduce action

Shift/reduce conflicts

Example:

\[
E ::= E + T | T
\]

A state:

\[
E ::= E \cdot + T \\
E ::= -T
\]

Can shift +
Can reduce \(E ::= T\)

Another example:

\[
S ::= \text{if } E \text{ then } S | \\
\quad \text{if } E \text{ then } S \text{ else } S | \cdots
\]

State:

\[
S ::= \text{if } E \text{ then } S \\
S ::= \text{if } E \text{ then } S \cdot \text{else } S
\]

Can shift else
Can reduce \(S ::= \text{if } E \text{ then } S\)
Avoiding shift/reduce conflicts

Can add lookahead to action table
  • fixes expression grammar conflict
  • won’t fix conflicts due to ambiguities, e.g. if/else

Can resolve in favor of shifting
  • tries to find longest r.h.s. before reducing
  • works well in practice, e.g. if/else
  • yacc, cup, et al. do this

Can rewrite grammar to remove conflict
  • E.g. MatchedStmt vs. UnmatchedStmt
  • E.g. change language by adding end

Reduce/reduce conflicts

Example:
Stmt ::= Type id ; | LHS = Expr ; | ...
...
LHS ::= id | LHS [ Expr ] | ...
...
Type ::= id | Type [ ] | ...

State:
Type ::= id .
LHS ::= id .

Can reduce Type ::= id
Can reduce LHS ::= id

Avoiding reduce/reduce conflicts

Can rewrite grammar to remove conflict
  • can be hard
    • e.g. C/C++ declaration vs. expression problem
    • e.g. MiniJava array declaration vs. array assignment problem

Can resolve in favor of one of the reduce actions
  • unlike shift/reduce, no good way to choose
  • yacc, cup, et al. pick reduce action for production listed textually first in specification

ASTs

The parser’s output is an abstract syntax tree (AST) representing the grammatical structure of the parsed input.

ASTs represent only semantically meaningful aspects of input program, unlike concrete syntax trees which record the complete textual form of the input program
  • no need to record keywords or punctuation like ( ), ; else
  • rest of compiler only cares about abstract structure
AST node classes

Each node in an AST is an instance of an AST class
- IfStmt, AssignStmt, AddExpr, VarDecl, etc.

Each AST class declares its own instance variables
holding its AST subtrees
- IfStmt has testExpr, thenStmt, and elseStmt
- AssignStmt has lhsAssignableExpr and rhsExpr
- AddExpr has arg1Expr and arg2Expr
- VarDecl has typeExpr and varName

AST class hierarchy

AST classes organized into an inheritance hierarchy based on
commonalities of meaning and structure

Each "abstract non-terminal" that has multiple alternative
concrete forms will have an abstract class that's the
superclass of the various alternative forms
-Stmt is abstract superclass of IfStmt, AssignStmt, etc.
-Expr is abstract superclass of AddExpr, VarExpr, etc.
-Type is abstract superclass of IntType, ClassType, etc.

AST extensions in project

New variable declarations:
- StaticVarDecl

New types:
- DoubleType
- ArrayType

New expressions:
- DoubleLiteralExpr
- OrExpr
- ArrayIndexExpr
- ArrayLengthExpr
- ArrayNewExpr

New/changed statements:
- IfStmt can omit else branch
- ForStmt
- BreakStmt
- AssignStmt can have ArrayIndexExpr as l.h.s.

Automatic parser generation in MiniJava

We use the CUP tool to automatically create a parser from a
specification file, Parser/mini.java.cup

The MiniJava Makefile automatically rebuilds the parser
whenever its specification file changes

A CUP file has several sections:
- introductory declarations included with the generated
  parser
- declarations of the terminals and nonterminals with their
types
  - the AST node or other value returned when finished parsing that
    nonterminal or terminal
- precedence declarations
- productions + actions
Terminal and nonterminal declarations

Terminal declarations we saw before:
/* reserved words: */
terminal CLASS, PUBLIC, STATIC, EXTENDS;
...
/* tokens with values: */
terminal String IDENTIFIER;
terminal Integer INT_LITERAL;

Nonterminals are similar:
nonterminal Program Program;
nonterminal List<RegularClassDecl> ClassDecls;
nonterminal RegularClassDecl ClassDecl;
...
nonterminal List<Stmt> Stmts;
nonterminal Stmt Stmt;
nonterminal List<Expr> Exprs, MoreExprs;
nonterminal Expr Expr, BaseExpr, AtomicExpr;
nonterminal String Identifier;

(Actually, use List<Stmt> in place of List<Stmt>, etc., since CUP doesn’t handle Java 1.5 generics directly)

Precedence declarations

Can specify precedence and associativity of operators
• equal precedence in a single declaration
• lowest precedence textually first
• specify left, right, or nonassoc with each declaration

Examples:
preference left AND, AND;
preference nonassoc EQUALS,EQUALS,
EXCLAIM_EQUALS;
preference left LESS_THAN, LESS_EQUAL,
GREATER_EQUAL, GREATER_THAN;
preference left PLUS, MINUS;
preference left STAR, SLASH;
preference left EXCLAIM;
preference left PERIOD;

Productions

All of the form:
LHS ::= RHS1 (): Java code 1 :
| RHS2 (): Java code 2 :
| ...
| RHSn (): Java code n :

Can label symbols in RHS with :var suffix to refer to its result value in Java code
• varleft is set to line in input where var symbol was

E.g. (slightly more complicated in real minijava.cup):
Expr ::= Expr::arg1 PLUS Expr::arg2
{ RESULT = new AddExpr(
arg1,arg2,arg1left); ;
| INT_LITERAL::value
{ RESULT = new IntLiteralExpr(
value.intValue(),valueleft); ;
| Expr::rCvr PERIOD Identifier::message
OPEN_PAREN Exprs::args CLOSE_PAREN
{ RESULT = new MethodCallExpr(
rCvr,message,args,rCvrleft); ;
| ...

Error handling

How to handle syntax error?

Option 1: quit compilation
  + easy
  - inconvenient for programmer

Option 2: error recovery
  + 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!!
Panic mode error recovery

When find a syntax error, skip tokens until reach a “landmark”

- landmarks in MiniJava: ; , ) , }
- once a landmark is found, hope to have gotten back on track

In top-down parser, maintain set of landmark tokens as recursive descent proceeds

- landmarks selected from terminals later in production
- as parsing proceeds, set of landmarks will change, depending on the parsing context

In bottom-up parser, can add special error nonterminals, followed by landmarks

- if syntax error, then will skip tokens till see landmark, then reduce and continue normally

E.g.
Stmt ::= ... | error ; | { error }
Expr ::= ... | ( error )