Bottom up parsing

- Construct parse tree for input from leaves up
  - reducing a string of tokens to single start symbol by inverting productions
- Bottom-up parsing is more general than top-down parsing and just as efficient — generally preferred in practice

```
int * int + int          T ::= int
int * T + int            T ::= int * T
T + int                 T ::= int
T + T                   E ::= T
T + E                   E ::= T + E
```

Read the productions found by bottom-up parse bottom to top; this is a rightmost derivation!

“Shift-reduce” strategy

- read (“shift”) tokens until the right hand side of “correct” production has been seen
- reduce handle to nonterminal, then continue
- done when all input read and reduced to start nonterminal

```
xyzabcdef          A ::= bc.D
```

LR(k)

- LR(k) parsing
  - Left-to-right scan of input, rightmost derivation
  - k tokens of look ahead
- Strictly more general than LL(k)
  - Gets to look at whole right hand side of production before deciding what to do, not just first k tokens
  - Can handle left recursion and common prefixes
  - As efficient as any top-down parsing
- Complex to implement
  - Generally need automatic tools to construct parser from grammar

LR Parsing Tables

- Construct parsing tables implementing a FSA with a stack
  - rows: states of parser
  - columns: token(s) of lookahead
  - entries: action of parser
    - shift, goto state X
    - reduce production “X ::= 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 may build huge tables

Global Plan for LR(0) Parsing

- Goal: Set up the tables for parsing an LR(0) grammar
  - Add S’ --> S$ to the grammar, i.e. solve the problem for a new grammar with terminator
  - Compute parser states by starting with state 1 containing added production, S’ --> .S$
  - Form closures of states and shifting to complete diagram
  - Convert diagram to transition table for PDA
  - Step through parse using table and stack
LR(0) Parser Generation

Example grammar:

```
S' ::= S $  // always add this production
S ::= beep | { L }
L ::= S | L ; S
```

- 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

Initial item:

```
S' ::= . S $  // always add this production
```

Closure

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

Initial state (1):

```
S' ::= . S $  
S ::= . beep 
S ::= . { L }
```

LALR-Look Ahead LR

- LALR(k) algorithm has fewer states ==> smaller tables
  - less general than LR(k), but still good in practice
  - size of tables acceptable in practice
- k == 1 in practice
  - most parser generators, including yacc and jflex, are LALR(1)

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

- do closure to compute new state Initial state (1):
  - S' ::= S $  
  - S ::= beep 
  - S ::= { L }

Reducing States

- If state has lhs ::= rhs . item, then it has a reduce lhs ::= rhs action

Example:

```
S ::= beep .
```

Accepting Transitions

- If state has S' ::= ... . S item, then add transition labeled $ to the accept action

Example:

```
S' ::= S . S
```

No label; this state always reduces this production

- what if other items in this state shift, or accept?
- what if other items in this state reduce differently?
Rest of the States, Part 1

State (4): if shift beep, goto State (3)
State (4): if shift $, goto State (4)
State (4): if shift $, goto State (5)
State (4): if shift $, goto State (6)

State (5):
L := S.

State (6):
S ::= { L ; } L := L ; S
State (6): if shift }, goto State (7)
State (6): if shift $, goto State (8)

State (7):
S ::= { L ; } .

State (8):
S ::= . beep
L := L ; S
L := L ; S;

State (8): if shift beep, goto State (3)
State (8): if shift $, goto State (4)
State (8): if shift $, goto State (9)

State (9):
L := L ; S.

LR(0) State Diagram

Building Table of States & Transitions

Create a row for each state
Create a column for each terminal, non-terminal, and $.

For every "state (i): if shift X goto state (j)" transition:
- If X is a terminal, put "shift, goto j" action in row i, column X
- If X is a non-terminal, put "goto j" action in row i, column X

For every "state (i): if $ accept" transition:
- Put "accept" action in row i, column $.

For every "state (i): lhs ::= rhs." action:
- Put "reduce lhs ::= rhs" action in all columns of row i.

Example

Table of This Grammar

<table>
<thead>
<tr>
<th>State</th>
<th>{</th>
<th>} beep</th>
<th>;</th>
<th>S</th>
<th>L</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s,g4</td>
<td>s,g3</td>
<td>g2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>al</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>reduce S ::= beep</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s,g4</td>
<td>s,g3</td>
<td>g5</td>
<td>g6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>reduce L ::= S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s,g7</td>
<td>s,g8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>reduce S ::= { L ; }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s,g4</td>
<td>s,g3</td>
<td>g9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>reduce L ::= L ; S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problems In Shift-Reduce Parsing

Can write grammars that cannot be handled with shift-reduce parsing

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

LR(0) example:
\[ E ::= E + T \mid T \]
State:
\[ E ::= E . + T \]
\[ E ::= T . \]
- Can shift
- Can reduce \[ E ::= T \]

LR(k) example:
\[ S ::= \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \ldots \]
State:
\[ S ::= \text{if } E \text{ then } S . \]
\[ S ::= \text{if } E \text{ then } S . \text{ else } S \]
- Can shift else
- Can reduce \[ S ::= \text{if } E \text{ then } S \]

Avoiding Shift-Reduce Conflicts

Can rewrite grammar to remove conflict
- E.g. Matched Stmt vs. Unmatched Stmt

Can resolve in favor of shift action
- try to find longest r.h.s. before reducing
  works well in practice
  yacc, jflex, et al. do this

Reduce/Reduce Conflicts

Example:
\[ \text{Stmt ::= Type } \text{id ; } \mid \text{LHS = Expr ; } \mid \ldots \]
\[ \ldots \]
\[ \text{LHS ::= id } \mid \text{LHS [ Expr ] } \mid \ldots \]
\[ \ldots \]
\[ \text{Type ::= id } \mid \text{Type [ ] } \mid \ldots \]
State:
\[ \text{Type ::= id . } \]
\[ \text{LHS ::= id . } \]
Can reduce \[ \text{Type ::= id } \]
Can reduce \[ \text{LHS ::= id } \]

Avoid 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 store problem

Can resolve in favor of one of the reduce actions
- but which?
  - yacc, jflex, et al. Pick reduce action for production listed textually first in specification

Abstract Syntax Trees

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
  - There’s no need to record keywords or punctuation like ()::, else
  - The rest of compiler only cares about the 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 subtree
- IfStmt has testExpr, thenStmt, and elseStmt
- AssignStmt has lhsVar and rhsExpr
- AddExpr has arg1Expr and arg2Expr
- VarDecl has typeExpr and varName

AST Class Hierarchy

AST classes are 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 For Project

New variable declarations:
- StaticVarDecl

New types:
- DoubleType
- ArrayType

New/changed statements:
- IfStmt can omit else branch
- ForStmt
- BreakStmt
- ArrayAssignStmt

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

Automatic Parser Generation in MiniJava

We use the CUP tool to automatically create a parser from a specification file, Parser/minijava.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:
```java
/* reserved words: */
terminal CLASS, PUBLIC, STATIC, EXTENDS;
... /* tokens with values: */
terminal String IDENTIFIER;
```

Nonterminals are similar:
```java
nonterminal Program Program;
nonterminal MainClassDecl MainClassDecl;
nonterminal List/<...>/ ClassDecls;
nonterminal RegularClassDecl ClassDecl;
...
nonterminal List/<Stmt>/; Stmts;
nonterminal Stmt Stmt;
nonterminal List/<Expr>/; Exprs;
```

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:
```java
precedence left AND_AND;
precedence nonassoc EQUALS_EQUALS,
EXCLAIM_EQUALS;
precedence left LESSTHAN, LESSEQUAL,
GREATEREQUAL, GREATERTHAN;
precedence left PLUS, MINUS;
precedence left STAR, SLASH;
prescendence left EXCLAIM;
prescendence left PERIOD;
```
Productions

All of the form:

\[ \text{LHS ::= RHS1 {: Java code 1 :}} \]
\[ | \text{RHS2 {: Java code 2 :}} \]
\[ | \cdots \]
\[ | \text{RHSn {: Java code n :}}; \]

Can label symbols in RHS with \texttt{var} suffix to refer to its result value in Java code

- \texttt{var} left is set to line in input where var symbol was

E.g.:

\[ \text{Expr ::= Expr:arg1 PLUS Expr:arg2 } \]
\[ (: \text{RESULT} = \text{new AddExpr( arg1, arg2, arg1left);}:) \]
\[ \text{| INT_LITERAL:val{: RESULT} = \text{new IntLiteralExpr(} \]
\[ \text{value.intValue(), valueright;}:) \]
\[ \text{| Expr:rcvr PERIOD Identifier:message OPEN_PAREN} \]
\[ \text{Exprs:args CLOSE_PAREN} \]
\[ (: \text{RESULT} = \text{new MethodCallExpr(} \]
\[ \text{rcvr, message, args, rcvrlert;}:) \]

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
- difficult to avoid streams of spurious errors

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

Panic Mode Error Recovery

When finding a syntax error, skip tokens until reaching 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 seeing landmark, then reduce and continue normally

E.g.:

\[ \text{Stmt ::= \ldots | error ; | \{ error \} } \]
\[ \text{Expr ::= \ldots | \{ error \} } \]

EBNF Syntax of initial MiniJava

Program ::= MainClassDecl { ClassDecl }
MainClassDecl ::= class ID { public static void main ( String [ ] ID ) { ( Stmt | ) } }
ClassDecl ::= class ID [ extends ID ] ( [ ClassVarDecl | MethodDecl | ] }
ClassVarDecl ::= ID Type ID { Formal [ , Formal | ] }
MethodDecl ::= public Type ID ( [ Stmt | return Expr | ] }
Formal ::= Type ID
Type ::= int | boolean | ID

Initial miniJava [continued]

Stmt ::= Type ID ;
| ( [ Stmt ] )
| if ( Expr ) Stmt else Stmt
| while ( Expr ) Stmt
| System.out.println ( Expr ) ;
| ID = Expr ;
Expr ::= Expr Op Expr
| ! Expr
| Expr - ID{ Expr | , Expr | | }
| ID | this
| Integer | true | false
| \{ Expr \}
Op ::= * | % | /
| < | <= | >= | > | == | != | &

CSE401 Au08 32

CSE401 Au08 33

CSE401 Au08 34

CSE401 Au08 35