**Shift-reduce parsing**

```
<table>
<thead>
<tr>
<th>derivation</th>
<th>stack</th>
<th>input stream</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+2+(3+4))+5 ← (1+2+(3+4))+5 shift</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1+2+(3+4))+5 ← (1+2+(3+4))+5 shift</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(E+2+(3+4))+5 ← (E) +2+(3+4))+5 reduce E→num</td>
<td>(E) +2+(3+4))+5</td>
<td>reduce E→E</td>
<td></td>
</tr>
<tr>
<td>(S+2+(3+4))+5 ← (S) +2+(3+4))+5 shift</td>
<td>(S) +2+(3+4))+5</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>(S+2+(3+4))+5 ← (S+) 2+(3+4))+5 shift</td>
<td>(S+) 2+(3+4))+5</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>(S+2+(3+4))+5 ← (S+2) (3+4))+5 reduce E→num</td>
<td>(S+2) (3+4))+5</td>
<td>reduce E→S+S+E</td>
<td></td>
</tr>
<tr>
<td>(S+E+(3+4))+5 ← (S+E) (3+4))+5 reduce S→S+E</td>
<td>(S+E) (3+4))+5</td>
<td>reduce S→S+E</td>
<td></td>
</tr>
<tr>
<td>(S+(3+4))5 ← (S+) (3+4))+5 shift</td>
<td>(S+) (3+4))+5</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>(S+(3+4))5 ← (S+) (3+4))+5 shift</td>
<td>(S+) (3+4))+5</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>(S+(3+4))5 ← (S+(3) +4))+5 reduce E→num</td>
<td>(S+(3) +4))+5</td>
<td>reduce E→num</td>
<td></td>
</tr>
</tbody>
</table>
```

**LR(0) states**

- A state is a set of items keeping track of progress on possible upcoming reductions
- An LR(o) item is a production from the language with a separator “.” somewhere in the RHS of the production
- Stuff before “.” is already on stack (beginnings of possible γ’s to be reduced)
- Stuff after “.” : what we might see next
- The prefixes α represented by state itself

**LR(k) parsing**

- As much power as possible out of parsing table with k look-ahead symbols
- LR(1) grammar = recognizable by a shift/reduce parser with 1 look-ahead.
- LR(1) item = LR(0) item + look-ahead symbols possibly following production
LR(1) state

• LR(1) state = set of LR(1) items
• LR(1) item = LR(0) item + set of lookahead symbols
• No two items in state have same production + dot configuration

LR(1) closure

Consider closure of item $A \rightarrow \beta \cdot C \delta \lambda$

Closure formed just as for LR(0) except

1. Lookahead symbols include characters following the non-terminal symbol to the right of dot: FIRST(δ)
2. If non-terminal symbol may produce last symbol of production (δ is nullable), lookahead symbols include lookahead symbols of production (λ)

LALR grammars

• Problem with LR(1): too many states
• LALR(1) (Look-Ahead LR)
  – Merge any two LR(1) states whose items are identical except for look-ahead
  – Results in smaller parser tables—works extremely well in practice
  – The usual technology for automatic parser generators
How are parsers written?

- Automatic parser generators: yacc, bison, CUP
- Accept LALR(1) grammar specification
  - plus: declarations of precedence, associativity
  - output: LR parser code (inc. parsing table)
- Some parser generators accept LL(1), e.g. javacc – less powerful, or LL(k), e.g. ANTLR
- Rest of this lecture: how to use parser generators
- Can we use parsers for programs other than compilers?

Classification of Grammars

SLR ⊇ LL(k) ⊇ LR(k) ⊇ LALR(1) ⊇ LR(1) ⊇ LR(0)

Associativity

\[
S \rightarrow S + E \mid E \\
E \rightarrow \text{num} \mid (S) \\
E \rightarrow E + E \mid \text{num} \mid (E)
\]

What happens if we run this grammar through LALR construction?

Conflict!

\[
E \rightarrow E + E \mid \text{num} \mid (E)
\]

\[
E \rightarrow E + E \ast \\
E \rightarrow E \ast + E +,\$
\]

shift/reduce conflict

\[
1+2+3
\]

shift: 1+(2+3)
reduce: (1+2)+3
Grammar in CUP

non terminal E; terminal PLUS, LPAREN...
precedence left PLUS;

“When shifting + conflicts with reducing
a production containing +, choose reduce”

E ::= E PLUS E
    |   LPAREN E RPAREN
    |   NUMBER ;

Precedence

• Also can handle operator precedence

E → E + E    |   T
T → T × T    |   num    |   ( E )

E ::= E PLUS E    |   E TIMES E    |   ...

Conflicts w/o precedence

E → E + E    |   E × E
    |   num    |   ( E )

Precedence in CUP

precedence left PLUS;
precedence left TIMES; // TIMES > PLUS
E ::= E PLUS E    |   E TIMES E    |   ...

Rule: in conflict, choose reduce if production
symbol higher precedence
than shifted symbol; choose shift if vice-versa
Summary

- Look-ahead information makes SLR(1), LALR(1), LR(1) grammars expressive
- Automatic parser generators support LALR(1)
- Precedence, associativity declarations simplify grammar writing
- Easiest and best way to read structured human-readable input

Compiler ‘main program’

class Compiler {
    void compile() throws CompileError {
        Lexer l = new Lexer(input);
        Parser p = new Parser(l);
        AST tree = p.parse();
        // calls l.getToken() to read tokens
        if (typeCheck(tree))
            IR = genIntermediateCode(tree);
            IR.emitCode();
    }
}

Thread of Control

Compiler.compile
   \ AST
   ^
Parser.parse
   | tokens
Lexer.getToken
   | bytes/chars
InputStream.read
   easier to make re-entrant

Semantic Analysis

Source code

lexical analysis
   tokens
   \ syntax errors
parsing
   abstract syntax tree
   \ semantic errors
semantic analysis
   \ valid programs: decorated AST
**Do we need an AST?**

- Old-style compilers: semantic actions generate code during parsing!
- Especially for stack machine:

  ```
  expr ::= expr PLUS expr
  { emitCode(add); }
  ```

Problems:
- hard to maintain
- limits language features (e.g., recursion)
- bad code!

**AST**

- **Abstract Syntax Tree** is a tree representation of the program. Used for
  - semantic analysis (type checking)
  - some optimization (e.g. constant folding)
  - intermediate code generation (sometimes intermediate code = AST with somewhat different set of nodes)
- Compiler phases = recursive tree traversals
- Object-oriented languages convenient for defining AST nodes

**Outline**

- Abstract syntax trees
- Type checking
- Symbol tables
- Using symbol tables for analysis

**Semantic Analysis**

Source code

- lexical analysis
  - tokens
  - lexical errors
- parsing
  - syntax errors
- abstract syntax tree
  - semantic errors
- semantic analysis
  - valid programs: decorated AST
Building the AST bottom-up

• Semantic actions are attached to grammar statements
• E.g. CUP: Java statement attached to each production
  non terminal Expr expr; ...
  expr ::= expr:e1 PLUS expr:e2
  { : RESULT = new Add(e1,e2); : }
• Semantic action executed when parser reduces a production
• Variable RESULT is value of non-terminal symbol being reduced (in yacc: $$)
• AST is built bottom-up along with parsing

Actions in S-R parser

non terminal Expr expr; ...  
expr ::= expr:e1 PLUS expr:e2  
{ : RESULT = new Add(e1,e2); : }
• Parser stack stores value of each non-terminal
  (1 + 2) + 3
  (1 + 2) + 3
  (E + 2) + 3
  (E + E) + 3
  (E) + 3
  E + 3
  RESULT = new Add(e1,e2)
  (E) + 3
  E + 3
  RESULT = e

How not to design an AST

• Introduce a tree node for every node in parse tree
  – not very abstract
  – creates a lot of useless nodes to be dealt with later

  S → E R  
  R → ε | + E R  
  E → num | ( S )

  (1 + 2) + 3

  S → E R  
  E → num | ( S )

  (1 + 2) + 3

How not to design the AST, part II

• Simple(minded) approach: have one class
  AST_node
• E.g. need information for if, while, +, *, ID, NUM
  class AST_node {
    int node_type;
    AST_node[] children;
    String name; int value; ...etc...
  }
• Problem: must have fields for every different kind of node with attributes
• Not extensible, Java type checking no help
Using class hierarchy

- Can use subclassing to solve problem
  - write abstract class for each “interesting” non-terminal in grammar
  - write non-abstract subclass for (almost) every prod’n

  \[ E \rightarrow E + E \mid E * E \mid -E \mid (E) \]

abstract class Expr {...} // E
class Add extends Expr { Expr left, right; ... }
class Mult extends Expr { Expr left, right; ... }
// or: class BinExpr extends Expr { Oper o; Expr l, r; }
class Negate extends Expr { Expr e; ... }

Creating the AST

non terminal Expr expr; ...

expr ::= expr:e1 PLUS expr:e2
{: RESULT = new BinaryExpr(plus, e1, e2); :}
| expr:e1 TIMES expr:e2
{: RESULT = new BinaryExpr(times, e1, e2); :}
| MINUS expr:e
{: RESULT = new UnaryExpr(negate, e); :}
| LPAREN expr:e RPAREN
{: RESULT = e; :}

Express plus, times, negate: Oper
UnaryExpr

Another Example

expr ::= num | (expr) | expr + expr | id
stmt ::= expr ; | if (expr) stmt |
         if (expr) stmt else stmt | id = expr ; |

abstract class Expr {...}
class Num extends Expr { Num(int value) ... }
class Add extends Expr { Add(Expr e1, Expr e2) ... }
class Id extends Expr { Id(String name) ... }
abstract class Stmt {...}
class If extends Stmt { If(Expr cond, Stmt s1, Stmt s2) }
class EmptyStmt extends Stmt { EmptyStmt() ... }
class Assign extends Stmt { Assign(String id, Expr e) ... }

And...top-down

- parse_X method for each non-terminal X
- Return type is abstract class for X

Stmt parseStmt() {
    switch (next_token) {
    case IF: eat(IF); eat(LPAREN);
          Expr e = parseExpr;
          eat(RPAREN);
          Stmt s2, s1 = parseStmt();
          if (next_token == ELSE) { eat(ELSE);
                                      s2 = parseStmt(); }
          else s2 = new EmptyStmt();
          return new IfStmt(e, s1, s2); }
    case ID: ...