1 Bottom-Up Parsing

- A bottom-up parser builds a derivation from the terminal symbols, working toward the start symbol.
- It consists of a *stack* and an *input*.
- Four actions:
  - *shift*, which pushes the next token onto the stack
  - *reduce*, removes Y1, ...,Yk, which are the right-hand side of some production X ::= Y1 ... Yk. From the top of the stack and replaces them by X.
  - *accept*, ends the parser with success.
  - *error*, ends the parser with an error message.

2 LALR(1) Parsing

- Question: How does the parser know, which action to invoke.
- Idea: Use a DFA applied to the *stack* to decide whether to shift or to reduce.
- The resulting parsers are called SLR, LR(0), LALR(1), LR(1) depending on the algorithm used to construct them.
- There is a trade off between accepted grammars and size of the automaton.
- LALR(1) is generally accepted as the best compromise.
- This is, what JavaCUP uses (also yacc, bison).
- All stronger methods have considerably larger automata.

3 Semantic Actions

- A parser usually does more than just recognize syntax.
- It could:
  - Evaluate code (simple interpreter)
  - Emit code (single pass compiler)
  - Build an internal data structure (multi pass compiler, interpreter)
- Generally, a parser performs *semantic actions*
- In a machine-generated bottom-up parser, they are added to the grammar submitted to the parser generator.
- There is a second stack, which keeps a value for each terminal or non-terminal. These can be used in the semantic action.
- In a recursive descent parser, semantic actions are embedded in the recognizer routines.

4 An Interpreter for Expressions

```
terminal PLUS, MINUS, TIMES, DIV, LPAREN, RPAREN;
terminal Integer NUMLIT;

non terminal Program;
non terminal Integer Expression, Term, Factor;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;

start with Program;
```

5 An Interpreter for Expressions (2)

```
Program    ::= Expression:e
               {: System.out.println(e.intValue()); :}
           ;
Expression ::= Expression:e PLUS Term:t
               {: RESULT = new Integer(e.intValue() + t.intValue()); :}
           |   Expression:e MINUS Term:t
               {: RESULT = new Integer(e.intValue() - t.intValue()); :}
           |   Term:t
               {: RESULT = t; :}
           ;
```


6 An Interpreter for Expressions (3)

```
Term   ::= Term:t TIMES Factor:f
           {: RESULT = new Integer(t.intValue() * f.intValue()); :}
       |   Term:t DIV Factor:f
           {: RESULT = new Integer(t.intValue() / f.intValue()); :}
       |   Factor:f
           {: RESULT = f; :}
       ;
Factor ::= NUMLIT:n
           {: RESULT = n; :}
       |   LPAREN Expression:e RPAREN
           {: RESULT = e; :}
       ;
```


7 Error Recovery

- After an error, the parser should be able to continue processing.
- Processing is for finding other errors, not for code generation or interpretation. These get disabled after the first error.
- Question: How can the parser recover from an error and resume normal parsing?

8 Error Recovery in Bottom-Up

- There are different schemes. The following is implemented in JavaCUP, yacc, bison.
- Introduce a special symbol error.
- The author of a parser can use error in productions.
- For instance:

```
Statement ::= Assignment
          |   IfStatement
          |   error ";"
          ;
```

9 Error Recovery in Bottom-Up (2)

- If the parser encounters an error, it will pop the stack until it gets into a state, where error is legal.
- At this point it shifts error onto the stack.
- Then, the input tokens are skipped, until the next input token is one that can legally follow the new state.
- This scheme is very dependent on a good choice of error productions.
- Assume a production Statement = error ";"
  - The parser encounters error inside a statement. It will pop the stack until it expects a statement.
  - At this point it shifts error onto the stack.
  - Then, the input tokens are skipped, until ";" is found.

10 Where to put error

- Different people recommend different things.
- It is a good idea to have a terminal after error to ensure termination.
- Examples:

```
Statement  ::= error SEMI
             |  LBRACE error RBRACE
             ;
Expression ::= LPAREN error RPAREN
             ;
```

- The generated parser will tell you the exact position of the error.