1 Part IV: Parsing

- Bottom-Up Parsing
- Parsing with JavaCUP
- Top-Down Parsing
- Error-Recovery

2 Scanners and Parsers

- Most compilers in practice have both a scanner for the lexical syntax and a parser for the context-free syntax.
- better modularity
- separation of concerns
- Characters \Rightarrow Scanner \Rightarrow Tokens
- Tokens \Rightarrow Parser \Rightarrow Syntax-Tree

 $\mathbf{2}$

3 From EBNF to BNF

For building parsers (especially bottom-up) a BNF grammar is often better, than EBNF. But it's easy to convert an EBNF Grammar to BNF:

- Convert every repetition { E } to a fresh non-terminal X and add $X = \epsilon \mid X E$.
- Convert every option [E] to a fresh non-terminal X and add $X = \epsilon \mid E$.

(We can convert X = A [E] B. to X = A E B | A B.)

- $\bullet\,$ Convert every group (E) to a fresh non-terminal X and add X=E.
- We can even do away with alternatives by having several productions with the same non-terminal.

 $X = E \mid E'$. becomes X = E. X = E'.



4 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 $X = Y1 \dots Yk$. 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.
- Question: How does the parser know, which action to invoke.

5 Simple Answer: Operator Precedence

- Suitable for languages of the form
 Expression = Operand Operator Operand with operands of varying precedence and associativity.
- Principle (token is the next input token):

}

if (token is an operand) shift; else if (stack does not contain an operator) shift; else { top = (topmost operator of stack); if (precedence(top) < precedence(token)) shift; else if (precedence(top) > precedence(token)) reduce; else if (top and token are right associative) shift; else if (top and token are left associative) reduce; else error;

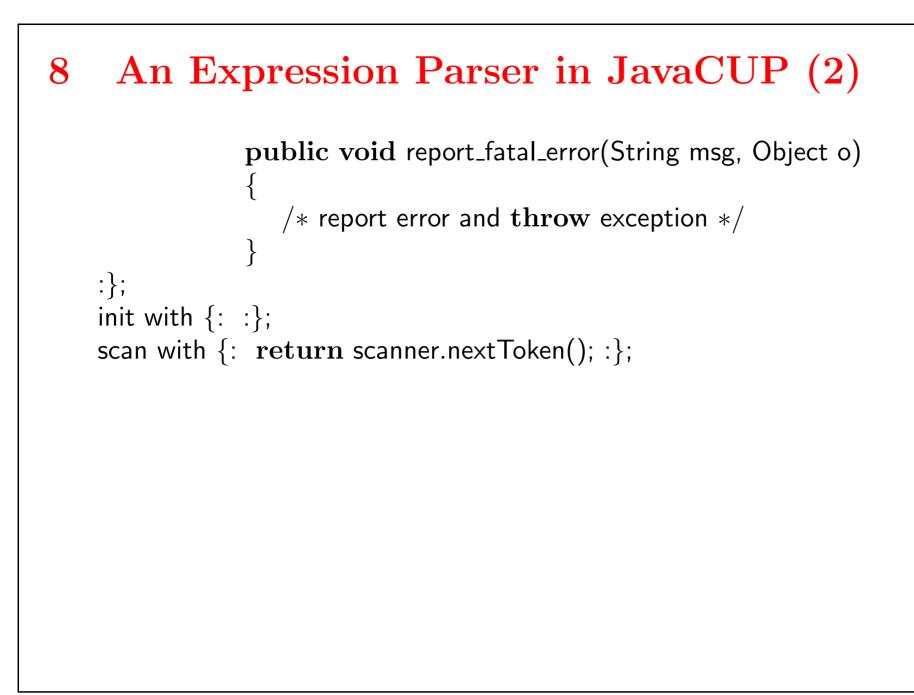
 $\mathbf{5}$

6 The Parser Generator JavaCUP

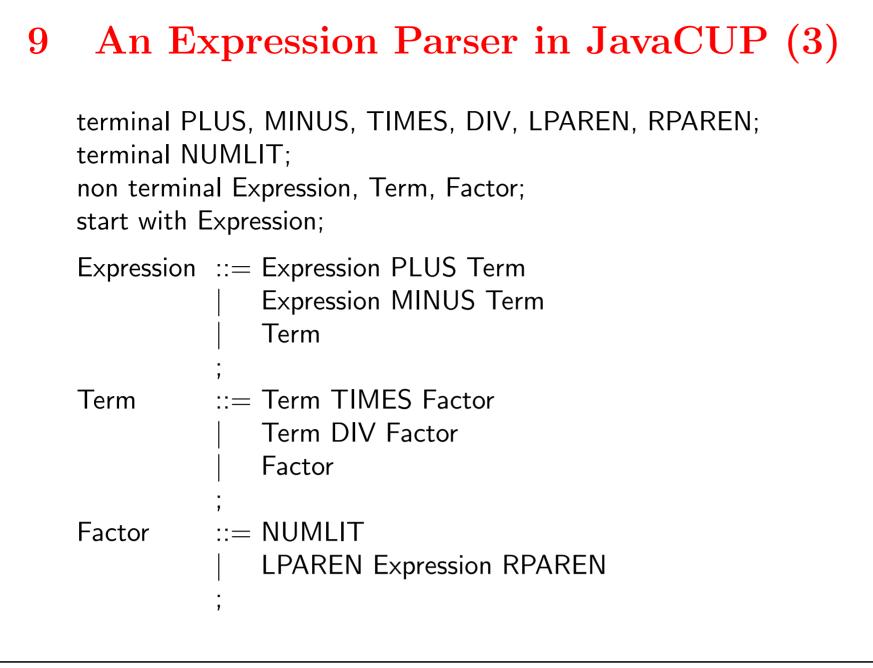
The original version is from http://www.cs.princeton.edu/ appel/modern/java/CUP/, but we use a local modified version.

- generates a class Parser.java, which contains the parser.
- generates a class Parser.tables, which contains the parsing tables.
- generates a class Tokens.java, which is suitable to be used by the scanner.
- if there are situations, where it wouldn't know, whether to shift or to reduce, it reports a conflict.

```
7 An Expression Parser in JavaCUP
package expression;
import jaco.framework.*;
action code {: :};
parser code {:
    Scanner scanner;
    public Parser(Scanner scanner)
    {
       this.scanner = scanner;
    }
    public void report_error(String msg, Object o)
    {
       /* report error */
    }
}
```









```
10 A shift-reduce Conflict
```

If we enter the grammar

 $\mathsf{Expression} \ ::= \mathsf{Expression} \ \mathsf{PLUS} \ \mathsf{Expression}$

without precedence JavaCUP will tell us:

```
*** Shift/Reduce conflict found in state #4
   between Expression ::= Expression PLUS Expression (*)
   and Expression ::= Expression (*) PLUS Expression
   under symbol PLUS
   Resolved in favor of shifting.
```

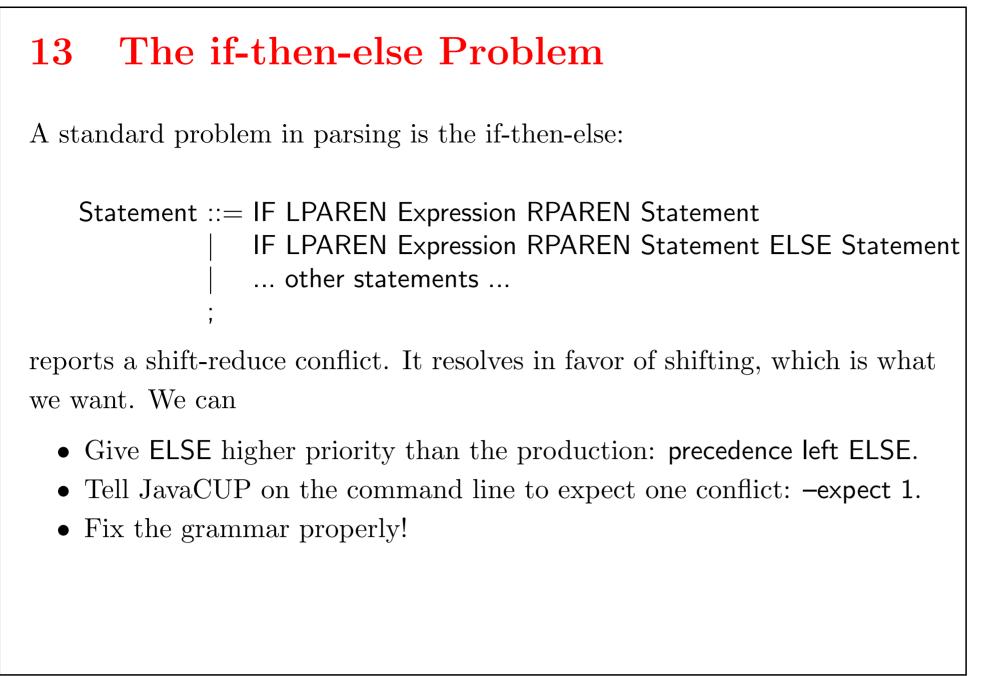
Telling JavaCUP that PLUS is left associative helps!

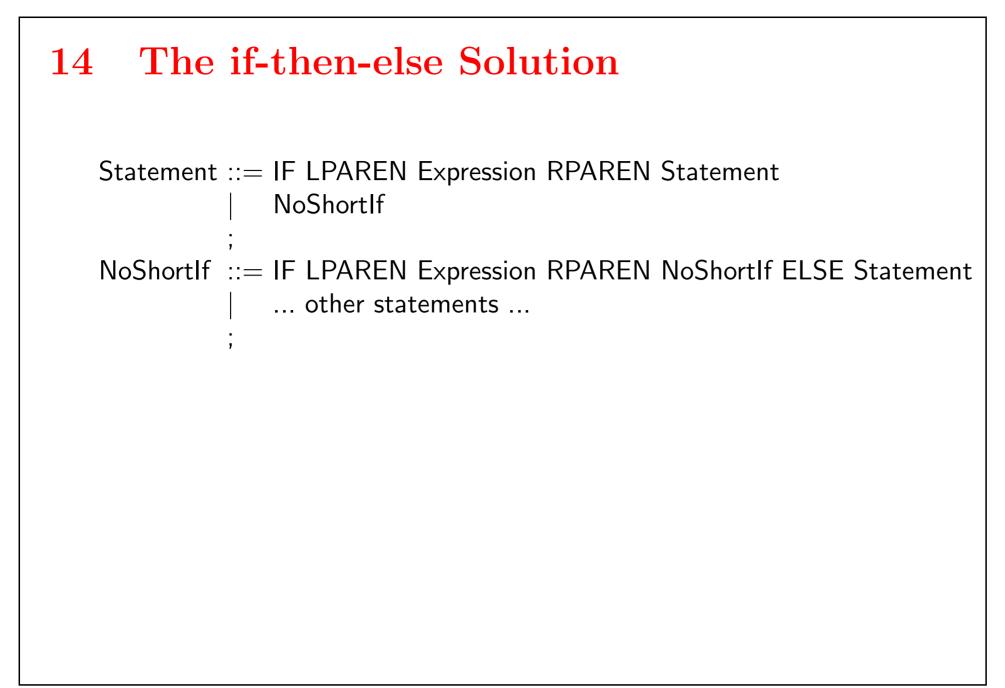


```
terminal PLUS, MINUS, TIMES, DIV, LPAREN, RPAREN;
terminal NUMLIT;
non terminal Expression, Term, Factor;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
start with Expression;
Expression ::= Expression PLUS Expression
| Expression MINUS Expression
| Expression TIMES Expression
| Expression DIV Expression
| NUMLIT
| LPAREN Expression RPAREN
;
```

12 Precedence

- A terminal has the given precedence (or lowest if unsepcified)
- a production has the precedence of its last terminal (lowest if unspecified, give if explicitly annotated).
- In a shift/reduce conflict
 - if the production has higher precedence reduce
 - if the terminal has higher precedence shift
 - if they are equal use associativity





15 A reduce-reduce Conflict

These conflicts are less common and often indicate a problem of the language rather than the grammar.

```
Expression ::= MExpression
| DExpression
;
MExpression:= IDENT TIMES IDENT
| IDENT
;
DExpression::= IDENT DIV IDENT
| IDENT
;
*** Reduce/Reduce conflict found in state #4
between MExpression ::= IDENT (*)
and DExpression ::= IDENT (*)
under symbols: {EOF}
Resolved in favor of the first production.
```

16 first(X), follow(X) and nullable

- first(X) are the terminals X can start with.
 - A terminal t is in first(X) if there is a parse tree, such that t is the leftmost leaf under X.
 - ϵ leaves do not count.
 - Example:

$$\begin{array}{l} \mathsf{A} = "b" \ "c" \ | \ \mathsf{B} \ "d". \\ \mathsf{B} = "a" \ | \ \epsilon. \\ \mathsf{first}(\mathsf{A}) = \{ \ \mathsf{b}, \ \mathsf{a}, \ \mathsf{d} \ \} \end{array}$$

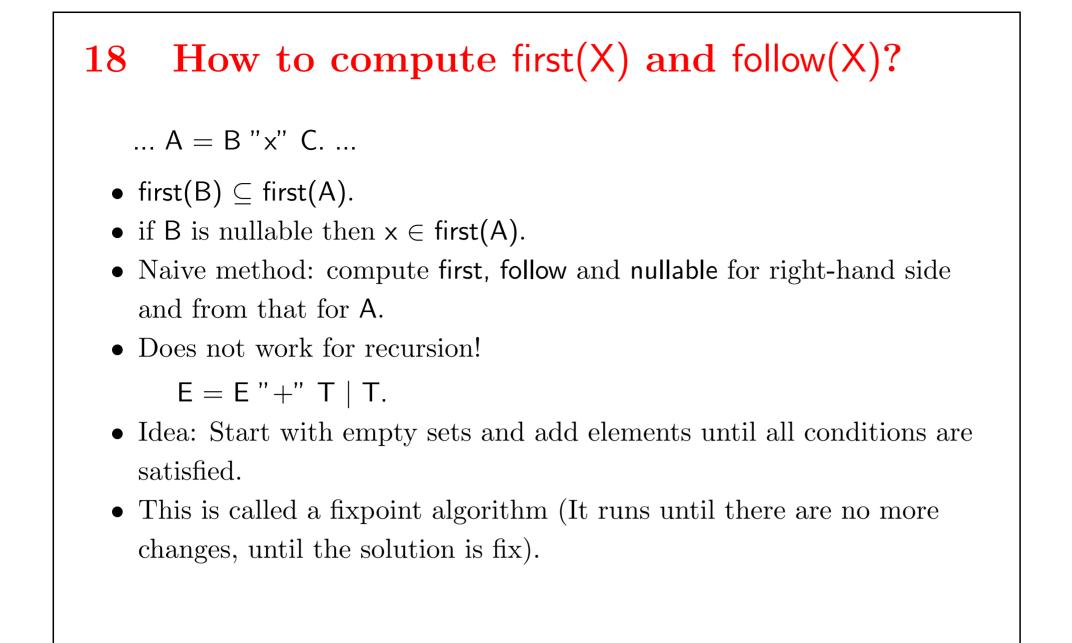
- follow(X) are terminals which can follow X.
 - A terminal t is in follow(X) if there is a parse tree such that t is the leftmost leaf after the leaves under X
 - Again, ϵ leaves do not count.
 - Example: follow(B) = { d }
- A non-terminal is nullable if it can derive the empty string (it may have only ϵ -leaves (Example: B is nullable)

¹⁶

17 Exercise

$$S = E$$
\$.
 $E = T$ "+" $E | T$.
 $T =$ "x".

Find the first and follow sets for T and $\mathsf{E}.$ Are there any nullable non-terminals?



Formal Definition: first(X), follow(X), nullable

 $\mathsf{first}(\mathsf{X}),\,\mathsf{follow}(\mathsf{X})$ and $\mathsf{nullable}$ are the smallest sets with the following properties:

For each production
$$X = Y1 ... Yk$$
, $1 \le i$, $j \le k$:
if { Y1, ..., Yk } \subseteq nullable
X \in nullable
if { Y1, ..., Yi-1 } \subseteq nullable
first(X) = first(X) \cup first(Yi)
if { Yi+1, ..., Yk } \subseteq nullable
follow(Yi) = follow(Yi) \cup follow(X)
if { Yi+1, ..., Yj-1 } \subseteq nullable
follow(Yi) = follow(Yi) \cup first(Yj)

20 Algorithm for first(X), follow(X) and nullable

```
\label{eq:relation} \begin{array}{l} \mbox{nullable} = \emptyset; \\ \mbox{for each terminal t } \{ \mbox{first}(t) = t; \mbox{follow}(t) = \emptyset; \} \\ \mbox{for each nonterminal Y } \{ \mbox{first}(Y) = \emptyset; \mbox{follow}(Y) = \emptyset; \} \\ \mbox{repeat } \{ \\ \mbox{nullable}' = \mbox{nullable}; \mbox{first}' = \mbox{first}; \mbox{follow}' = \mbox{follow}; \\ \mbox{for each production } X = Y1 \ ... \ Yk, \ 1 \leq i, \ j \leq k \; \{ \\ \mbox{if } \{ \ Y1, \ ..., \ Yk \; \} \subseteq \mbox{nullable} \\ \mbox{nullable} = \mbox{nullable} \ UX; \\ \mbox{if } \{ \ Y1, \ ..., \ Yi-1 \; \} \subseteq \mbox{nullable} \\ \mbox{first}(X) = \mbox{first}(X) \ \cup \mbox{first}(Yi); \\ \mbox{if } \{ \ Yi+1, \ ..., \ Yk \; \} \subseteq \mbox{nullable} \\ \mbox{follow}(Yi) = \mbox{follow}(Yi) \ \cup \mbox{follow}(X); \\ \mbox{if } \{ \ Yi+1, \ ..., \ Yj-1 \; \} \subseteq \mbox{nullable} \\ \mbox{follow}(Yi) = \mbox{follow}(Yi) \ \cup \mbox{follow}(Yj); \\ \\ \mbox{lumbda} \} \\ \mbox{until (nullable} = \mbox{nullable}', \mbox{first} = \mbox{first}', \mbox{follow} = \mbox{follow}'); \\ \end{array}
```

21 LR(0) Parsing

- Idea: Use a DFA applied to the *stack* to decide whether to shift or to reduce.
- The states of the DFA are sets of LR(0) items.
- An LR(0) item is of the form [X = A _ B], where X is a non-terminal and A,B are strings of terminals and non-terminals (possibly empty).
- An LR(0) item describes a possible situation during parsing, where
 - X=AB. is a production, which is currently possible.
 - A is on the stack.
 - B is in the input.
 - the _ describes the border between stack and input.
- Example: $[E = T_{-}"+"E]$

22 LR(0) Parsing (2)

• Principle:

- *shift*, in a state where $[X = A _ b B]$ if the next token is b.
- reduce, in a state $[X = A_{-}]$
- The resulting parser is called LR(0), since it parses left-to-right, describes a rightmost derivation. The 0 means, that the parser uses no lookahead on the input.

23 SLR Parsing

- Problem: Some states contain shift and reduce items.
- Example:

• LR(0) state construction gives a state containing the items

$$\begin{bmatrix} \mathsf{E} = \mathsf{T}_{-}" + " & \mathsf{E} \end{bmatrix}$$
$$\begin{bmatrix} \mathsf{E} = \mathsf{T}_{-} \end{bmatrix}$$

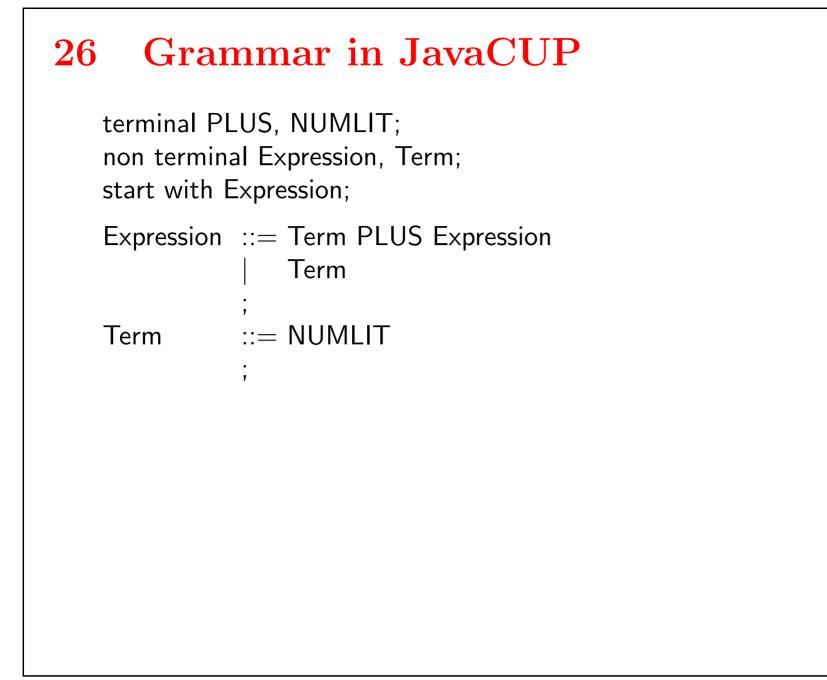
- If we see "+" as the next input token should we shift or reduce?
- Solution: Reduce only if the symbol is in follow(E).
- The resulting parser is called *simple LR* or SLR.
- The number of states is the same as in LR(0).

24 LALR(1) Parsing

- Sometimes, in specific states not all terminals from follow(X) can really occur.
- Idea: Propagate state-specific follow symbols.
- Reduce only if the symbol is in the state specific follow symbols.
- The resulting parser is called LALR(1) for Look-Ahead-LR.
- The number of states is the same as in LR(0) and SLR.
- This is, what JavaCUP uses (also yacc, bison).
- If an LALR(1) parser generator gives a conflict, then for all practical purposes it cannot know, what to do in certain situations.

25 LR(1) Parsing

- LR(1) parsing refines the notion of state. A state is now a set of LR(1) items, where each item is of the form [X = A _ B ; c] and c is a terminal.
 - X=AB. is a production, which is currently possible.
 - A is on the stack.
 - B c is in the input.
 - $\bullet\,$ the $_$ describes the border between stack and input.
- The rest of the construction is similar to LR(0), except that we reduce in a state with item [X = A _ ; c] only if the next input token is c.
- The result is called LR(1) parsing, because now we use one token lookahead to make our decision.
- LR(1) parsers are slightly more powerful than LALR(1) parsers.
- But, there are many more LR(1) states than LR(0) states. Often we have a *state explosion*



```
27 States in JavaCUP
```

The option -dump_states yields the following output

```
START lalr_state [0]: {
    [Expression ::= (*) Term , {EOF }]
    [Expression ::= (*) Term PLUS Expression , {EOF }]
    [Term ::= (*) NUMLIT , {EOF PLUS }]
    [$START ::= (*) Expression EOF , {EOF }]
}
transition on Expression to state [3]
transition on NUMLIT to state [2]
transition on Term to state [1]
lalr_state [1]: {
    [Expression ::= Term (*) , {EOF }]
    [Expression ::= Term (*) PLUS Expression , {EOF }]
}
transition on PLUS to state [5]
```

```
lalr_state [2]: {
    [Term ::= NUMLIT (*) , {EOF PLUS }]
}
lalr_state [3]: {
    [$START ::= Expression (*) EOF , {EOF }]
}
transition on EOF to state [4]
lalr_state [4]: {
    [$START ::= Expression EOF (*) , {EOF }]
}
```

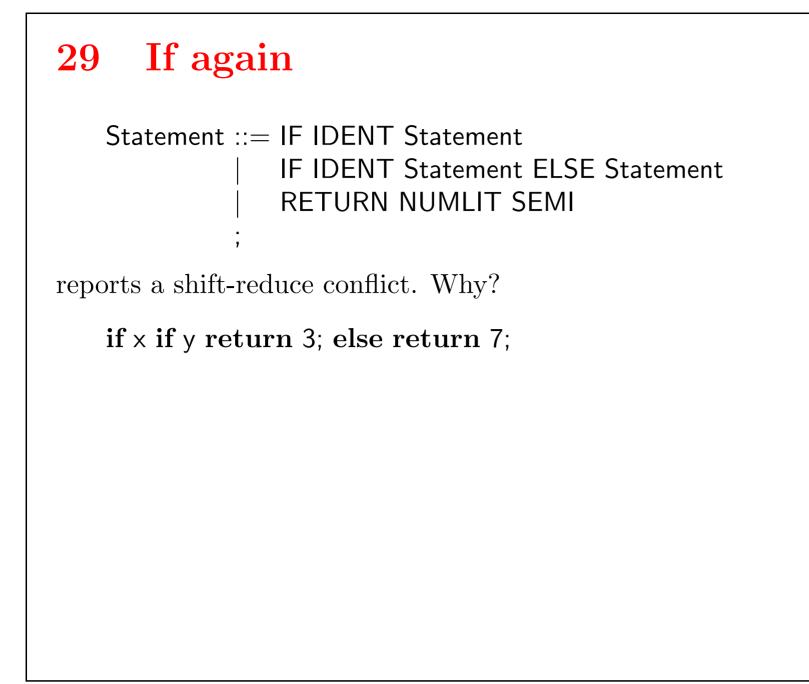
```
lalr_state [5]: {
    [Expression ::= (*) Term , {EOF }]
    [Expression ::= Term PLUS (*) Expression , {EOF }]
    [Expression ::= (*) Term PLUS Expression , {EOF }]
    [Term ::= (*) NUMLIT , {EOF PLUS }]
}
transition on Expression to state [6]
transition on NUMLIT to state [2]
transition on Term to state [1]
lalr_state [6]: {
    [Expression ::= Term PLUS Expression (*) , {EOF }]
}
```

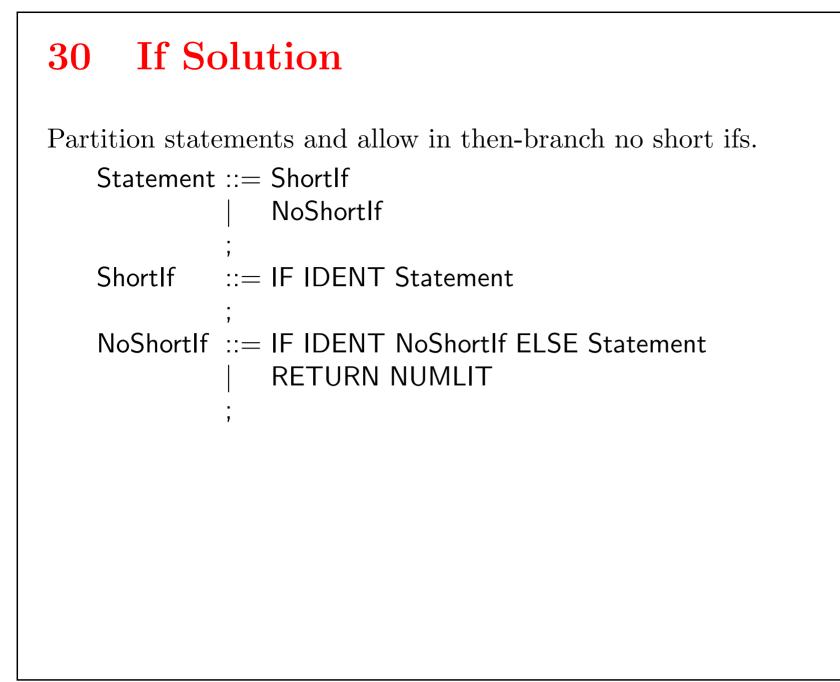
28 Debugging JavaCUP

Calling debug_parse() in ParserTest instead of parse() (You can use -debug), using input 5+3 yields

Initializing parser # Current Symbol is #8 # Shift under term #8 to state #2 # Current token is #2 # Reduce with prod #3 [NT=2, SZ=1] # Goto state #1 # Shift under term #2 to state #5 # Current token is #8 # Shift under term #8 to state #2 # Current token is #0

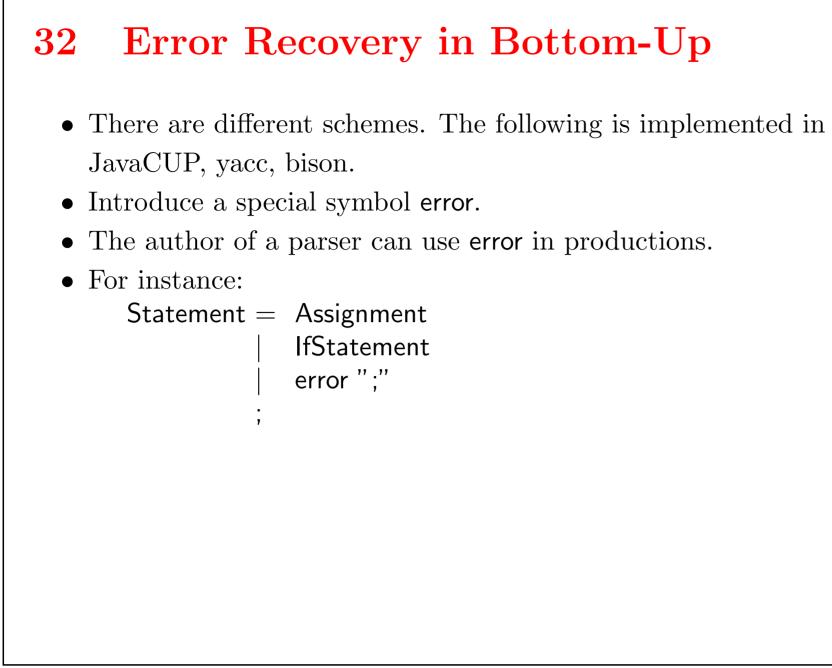
```
# Reduce with prod #3 [NT=2, SZ=1]
# Goto state #1
# Reduce with prod #2 [NT=1, SZ=1]
# Goto state #6
# Reduce with prod #1 [NT=1, SZ=3]
# Goto state #3
# Shift under term #0 to state #4
# Current token is #0
# Reduce with prod #0 [NT=0, SZ=2]
# Goto state #-1
```





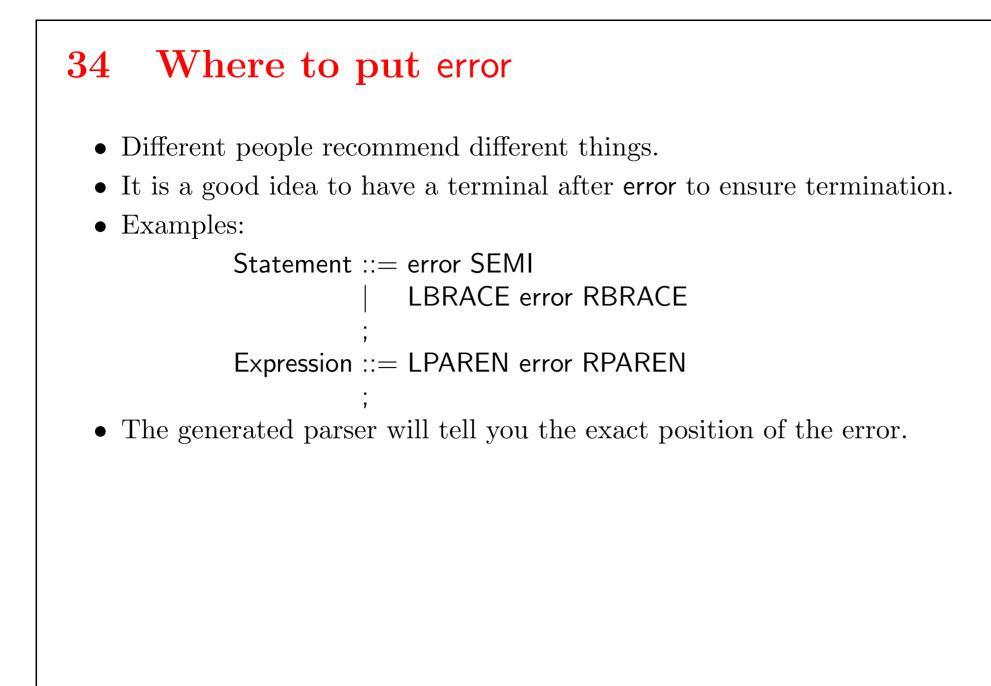
31 Error Recovery

- 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?



33 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.
 - 36



35 Semantic Actions

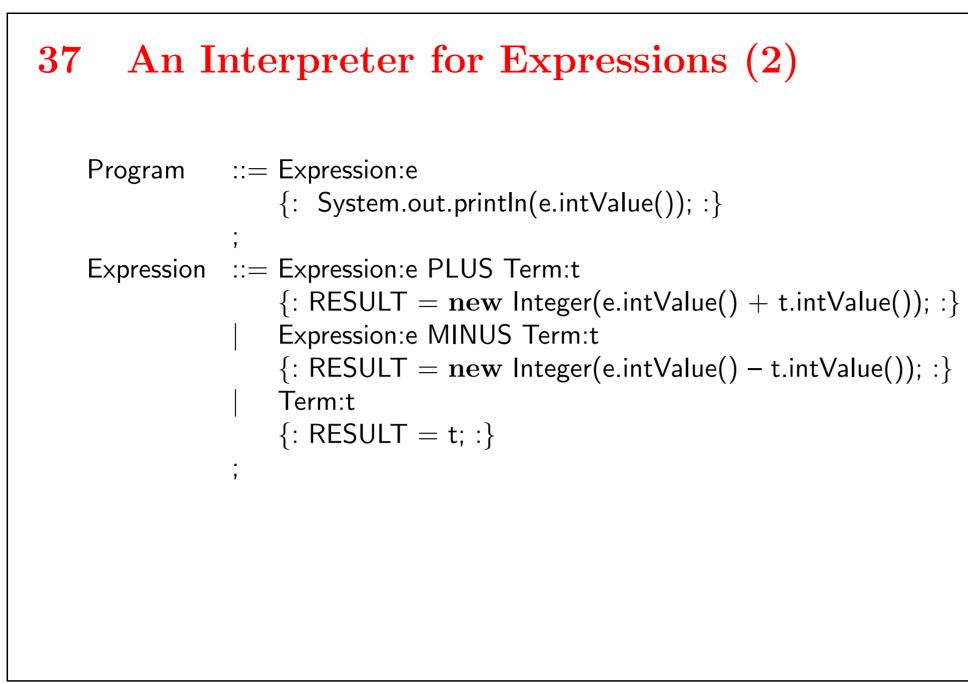
- A parser usually does more than just recognize syntax.
- It could:
 - Evaluate code (simple interpreter)
 - Emit code (single pass compiler)
 - Build an internal data structure (multi pass compiler, interpreter)
- Generally, a parser performs *semantic actions*
- In a machine-generated bottom-up parser, they are added to the grammar submitted to the parser generator.
- In a recursive descent parser, semantic actions are embedded in the recognizer routines.

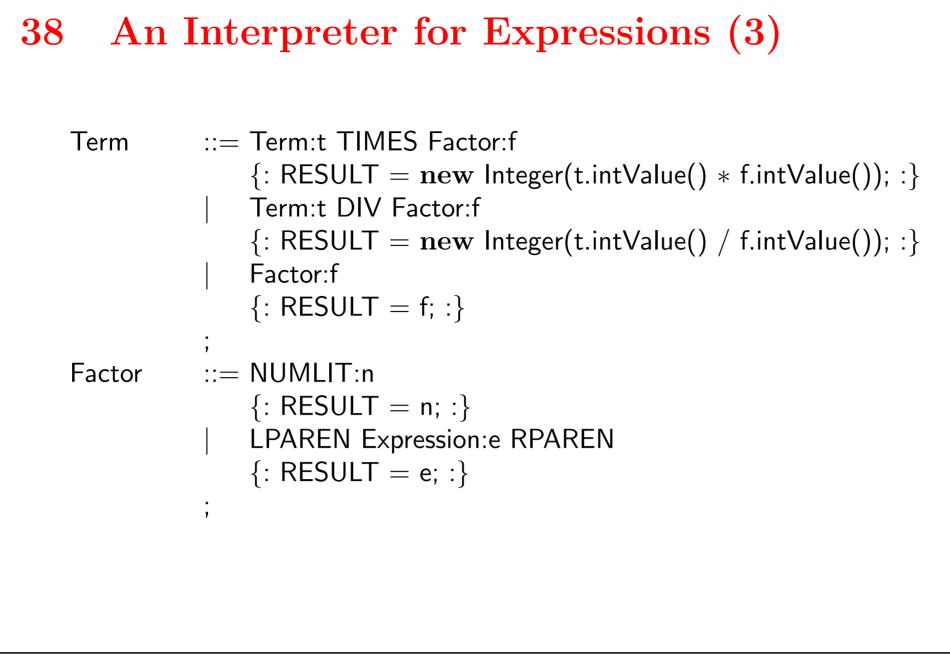
36 An Interpreter for Expressions

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;





39 Top-Down Parsing

- Regular languages are limited in that they cannot express nesting.
- Therefore, finite state machines cannot recognize context-free grammars.
- Let's try it anyway: A = ident A numlit | numlit. leads after simplification to the following parser:

```
void A() {
    if (token == IDENT) {
        nextToken();
        A();
        if (token == NUMLIT) nextToken(); else error();
    } else if (token == NUMLIT)
        nextToken();
    else
        error();
}
```

40 Deriving a Parser from EBNF

To derive a parser from a context-free grammar written in EBNF style:

- Introduce one function void A() for each non-terminal A
- The task of A() is to recognize sub-sentences derived from A, or issue an error if no A was found.
- Translate all regular expressions on the right-hand-side of productions as before, but
 - every occurrence of a non-terminal B maps to B()
 - Recursion in the grammar translates naturally to recursion in the parser.
- This technique of writing parsers is called parsing by *recursive descent* or *predictive parsing*.

41 A Parser for Expressions

```
42 Eliminating Left Recursion
Expression = Term { ( "-" | "+" ) Term }.
Term = Factor { ( "*" | "/" ) Factor }.
Factor = numlit | "(" Expression ")".
void Expression() {
    Term();
    while (token == MINUS || token == PLUS) {
        nextToken();
        Term();
    }
}
```

```
43 Another Problem
```

```
Factor = ident | ident "[" Expression "]" | numlit.
void Factor() {
    if (token == IDENT) {
        ??
    } else {
        if (token == NUMLIT)
            nextToken();
        else error();
    }
}
```

44 Left Factoring

```
Factor = ident [ "[" Expression "]" ] | numlit.
void Factor() {
     if (token == IDENT) {
           if (token == LBRACKET) {
                 nextToken();
                 Expression();
                 if (token == RBRACKET)
                       nextToken();
                 else error();
            }
     } else {
           if (token == NUMLIT)
                 nextToken();
           else error();
      }
}
```

45 LL(1) Grammar

- Definition: A simple BNF grammar is LL(1) if for all nonterminals X: if X appears on the left-hand side of two productions X=E1. and X=E2. then
 - first(E1) \cap first(E2) = \emptyset .
 - either (neither E1 nor E2 is nullable)
 - or (exactly one, say E1 is nullable and $first(X) \cap follow(E2) = \emptyset$.
- LL(1) stands for "left-to-right-parse, leftmost derivation, 1 symbol lookahead".
- Recursive descent parsers work only for LL(1) grammars.
- Elimination of left recursion and left-factoring work often, but not always.



• We choose a set of stop-symbols, e.g. } ;)

• If we encounter an error, we call skip(), give an error message and return.

- skip() skips the input to the next stop symbol.
- It also skips subblocks { ... } completely.
- We do not print two error messages for the same position.

```
a = 5 * (3 4);
```

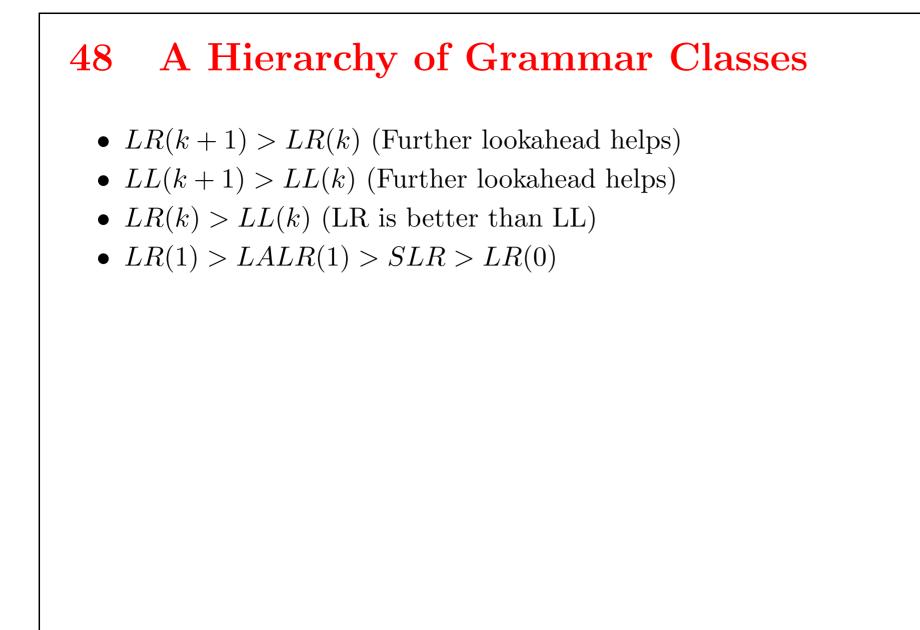
{

}



47 Summary Top-Down Parsing

- A context-free grammar can be converted directly into a program scheme for a recursive descent parser.
- A recursive-descent parser builds a derivation top down, from the start symbol towards the terminal symbols.
- Weakness: Must decide what to do based on first input token.
- This works only if the language is LL(1).





49 Top-Down / Bottom-Up

Top-Down

- + easy to write by hand.
- + flexible embedding in compiler possible.
- harder to maintain.
- error recovery can be tricky.
- deep recursion can be inefficient.

Bottom-Up

- + larger class of languages and grammar.
- needs tool to generate
- less flexible to embed in compiler
- depends on quality of tool

Mixtures are possible. Some commercial compilers use recursive descent, with operator precedence for expressions to get rid of deep recursion.