Compiler Theory

(Syntax Analysis – Parsing)

004

The role of syntax analysis

- For well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.
- Three general types
  - Universal (can parse any grammar)
  - Top-down
  - Bottom-up

Grammars for Exp

- We shall focus on expressions because these present more of a challenge because of,
  - Associativity of operators
  - Precedence of operators

- Statements that begin with for e.g. **While** are typically easier to parse, because the keyword guides the choice of which grammar rule to use to match the input.

A grammar to capture expressions

- E ->
  - E + T
  - T
- T ->
  - T * F
  - F
- F ->
  - (E)
  - id

- Note that this grammar cannot be parsed using a top down parse. Why ?
- But it is suitable to be parsed used a bottom-up parser.
- The next slide gives you an alternative grammar which can be parsed top-down.

A grammar to capture expressions (ii)

- E
  - T E'
- E'
  - + T E' | ε
- T
  - F T '
- T '
  - * F T ' | ε
- F
  - (E) | **id**

- Left recursion has been removed.
- The grammar (which is equivalent to the one before) can now be fed into a top-down parser.

Lexical versus Syntactic Analysis

- We know that everything that can be described by a regular grammar can be described by a context-free grammar … so you could ask why don’t we use context-free grammars to define lexical rules as well. Here are some reasons:
  - It is good practise to separate the syntactic structure of a language into lexical and non-lexical parts mainly for modularisation of the front-end
  - Lexical rules are normally quite simple to describe … and don’t require a notation as powerful and expressive as a CF grammar
  - Regular expressions generally provide a more concise and easy-to-understand notation
  - More efficient lexers can be constructed from RegExprs

Syntax-Error Handling (i)

- A compiler is expected to help the programmer in locating and tracking down errors !

- **Lexical** Errors
  - e.g. Misspellings of ids, keywords and missing quotes around text intended as a string

- **Syntactic** Errors
  - e.g. Misplaced semi-colons, extra or missing braces { }

Syntax-Error Handling (ii)

- A compiler is expected to help the programmer in locating and tracking down errors !
- **Semantic** Errors
  - e.g. Type mismatches between operator and operands.
- **Logical** Errors
  - e.g. Incorrect reasoning, = instead of ==. Program may be well-formed but not what the programmer wants.

Goals of Error-handler in Parser

- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to detect subsequent errors
- Add minimal overhead to the processing of correct programs
- The error handler should at least inform the programmer of the offending line in the source

Error-Recovery Strategies (i)

- *Simplest possible approach*
  - quit on first error
- *Panic-Mode Recovery*
  - Synchronizing tokens – upon discovering an error, the parser discards input symbols one at a time until a synch token is matched. e.h. ; or }
- *Phrase-Level Recovery*
  - When discovering an error the parser might try some local correction. e.g. Replace comma with semi-colon, insert or delete semi-colon, etc

Error-Recovery Strategies (ii)

- *Error Productions*
  - Tries to anticipate common errors and actually includes them in the grammar so that the parser generates appropriate error diagnostics about the erroneous construct. Not common.
- *Global Correction*
  - Tries to infer the closest correct program , however this is very expensive and not practical. Only of theoretical interest.

Derivations (i)

- E
  - E + E | E * E | -E | (E) | id

- A derivation of -(id) from E is the sequence of replacements

- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

Derivations – Leftmost (ii)

- Leftmost derivation
  - The leftmost non-terminal in each sentential is always chosen.

- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

Derivations – Rightmost (iii)

- Rightmost derivation
  - The rightmost non-terminal is always chosen.

- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- Now …. A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non-terminals. The parse tree (final step in derivation) on the next slide results from the derivation above and the one on the previous slide. The sequence however maps the LeftMost derivation.

- Each interior node represents the application of a production.

Sequence of Parse Trees for derivations

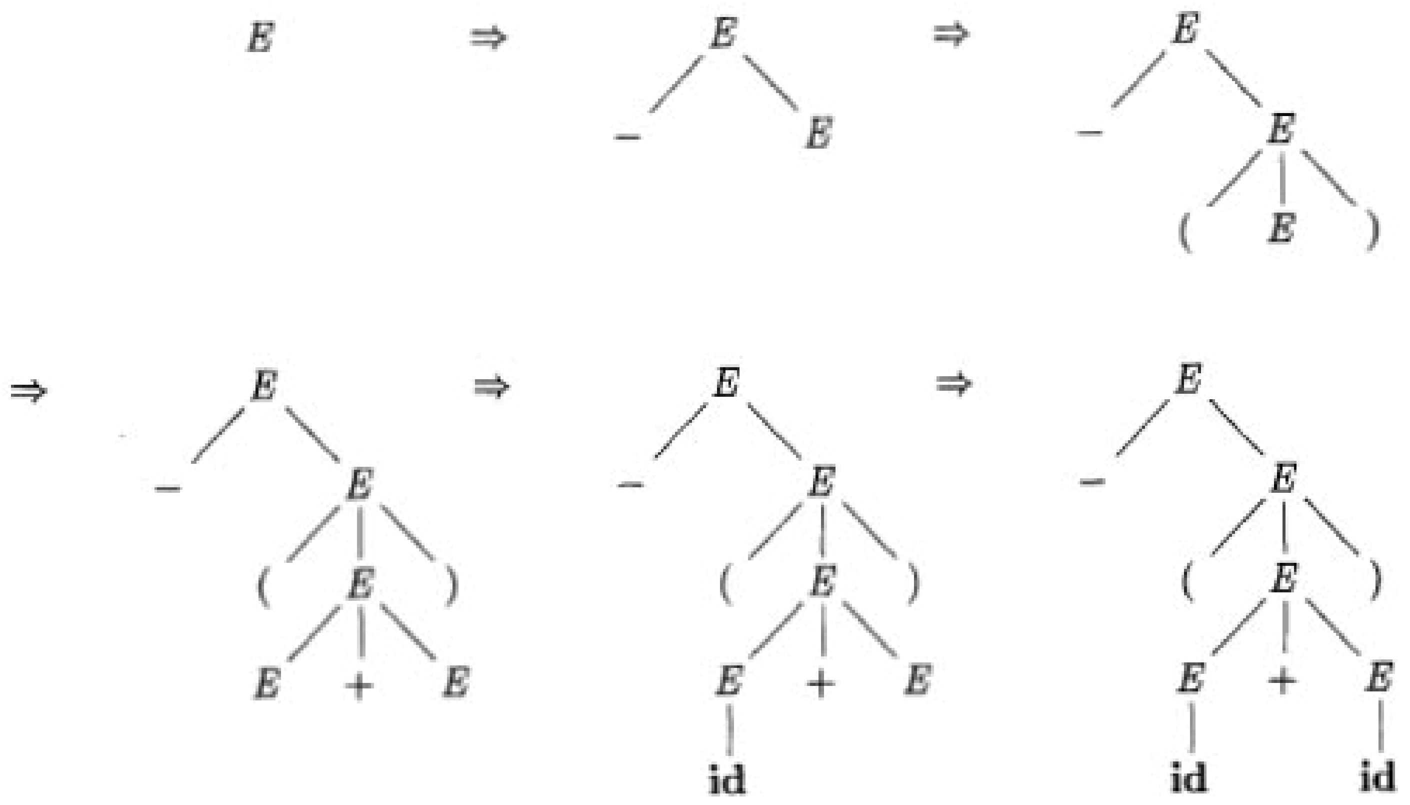


Figure 4.4: Sequence of parse trees for derivation (4.8)

Ambiguity (i), Two Leftmost derivations !!

- E => E + E
  => id + E
  => id + E * E
  => id + id * E
  => id + id * id

- E => E * E
  => E + E * E
  => id + E * E
  => id + id * E
  => id + id * id

Ambiguity (ii)

- An ambiguous grammar is one that produces more that one leftmost derivation or more than one rightmost derivation

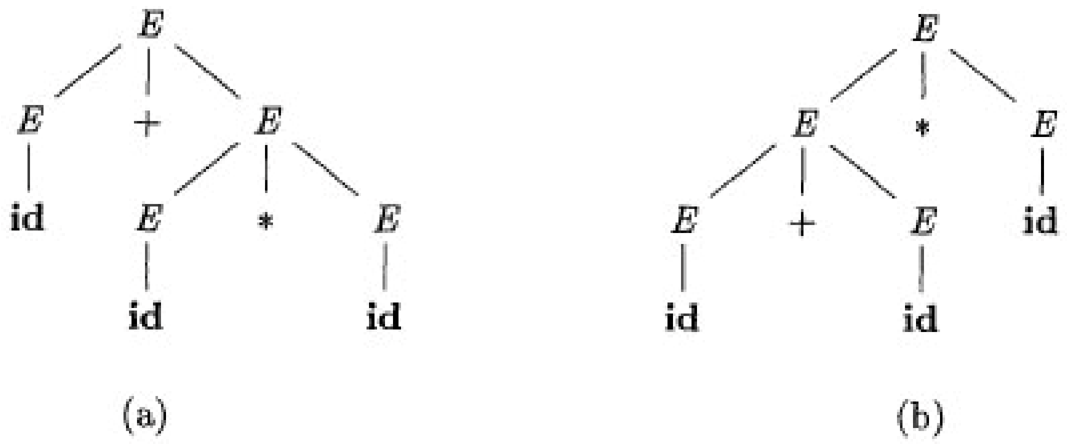


Figure 4.5: Two parse trees for **id+id*id**

Eliminating Ambiguity (i)

- Sometimes it is possible to eliminate ambiguity in grammars.

- *stmt*
  - **if** *expr* **then** *stmt*
  - **if** *expr* **then** *stmt* **else** *stmt*
  - **other**

If E_1 then if E_2 then S_1 else S_2 …

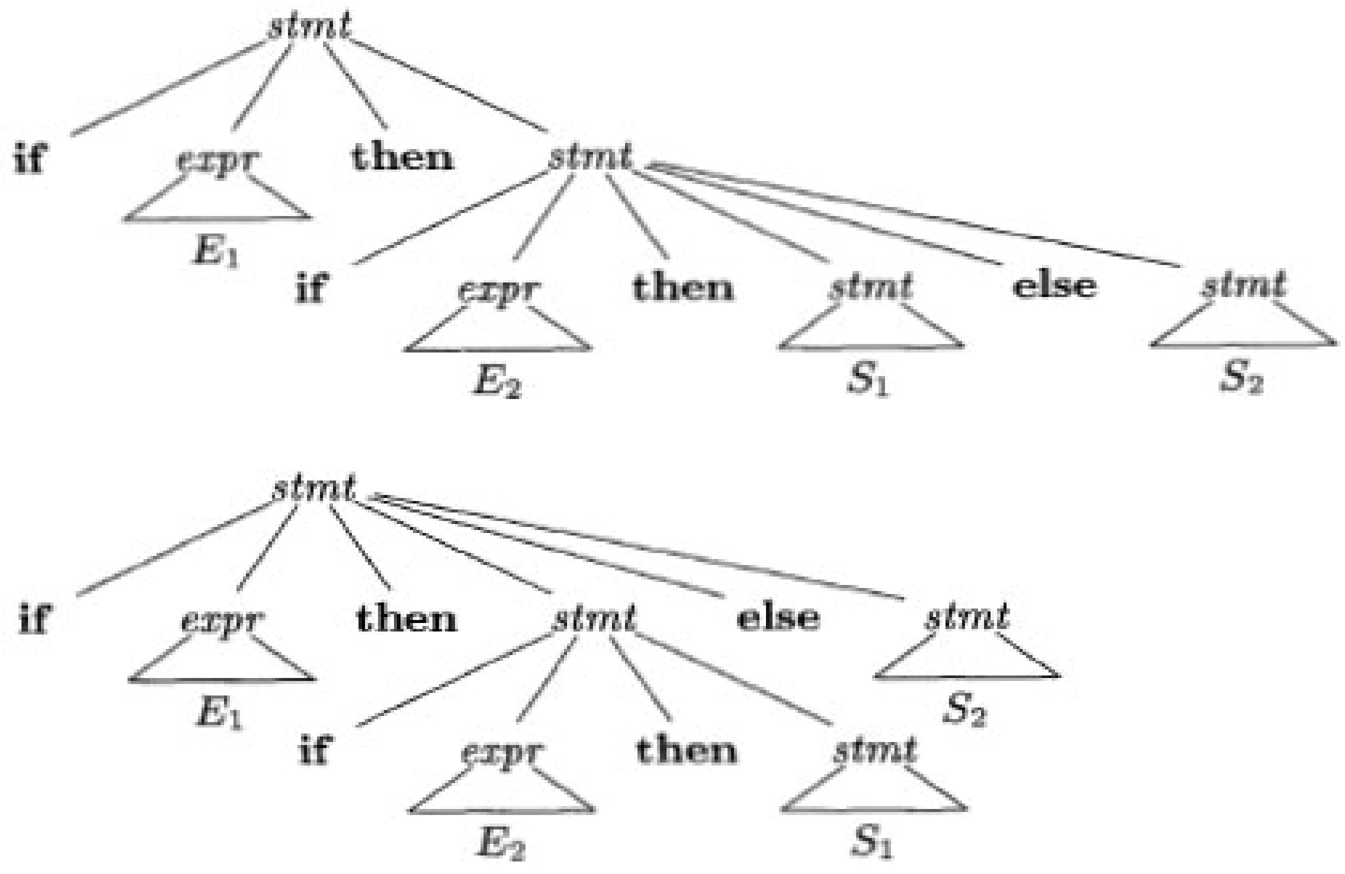


Figure 4.9: Two parse trees for an ambiguous sentence

Eliminating Ambiguity (iii)

- Problem here is the dangling else !!
- The idea is that a statement appearing between a then an and else must be matched.
- The grammar in the next slide makes sure that, for the 'if' statement in the previous slide, there is only one parse tree.

Eliminating Ambiguity (iv)

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{matched\_stmt} \\
 & | & \mathit{open\_stmt} \\
\mathit{matched\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{matched\_stmt} \\
 & | & \textbf{other} \\
\mathit{open\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{open\_stmt}
\end{array}
$$

Figure 4.10: Unambiguous grammar for if-then-else statements

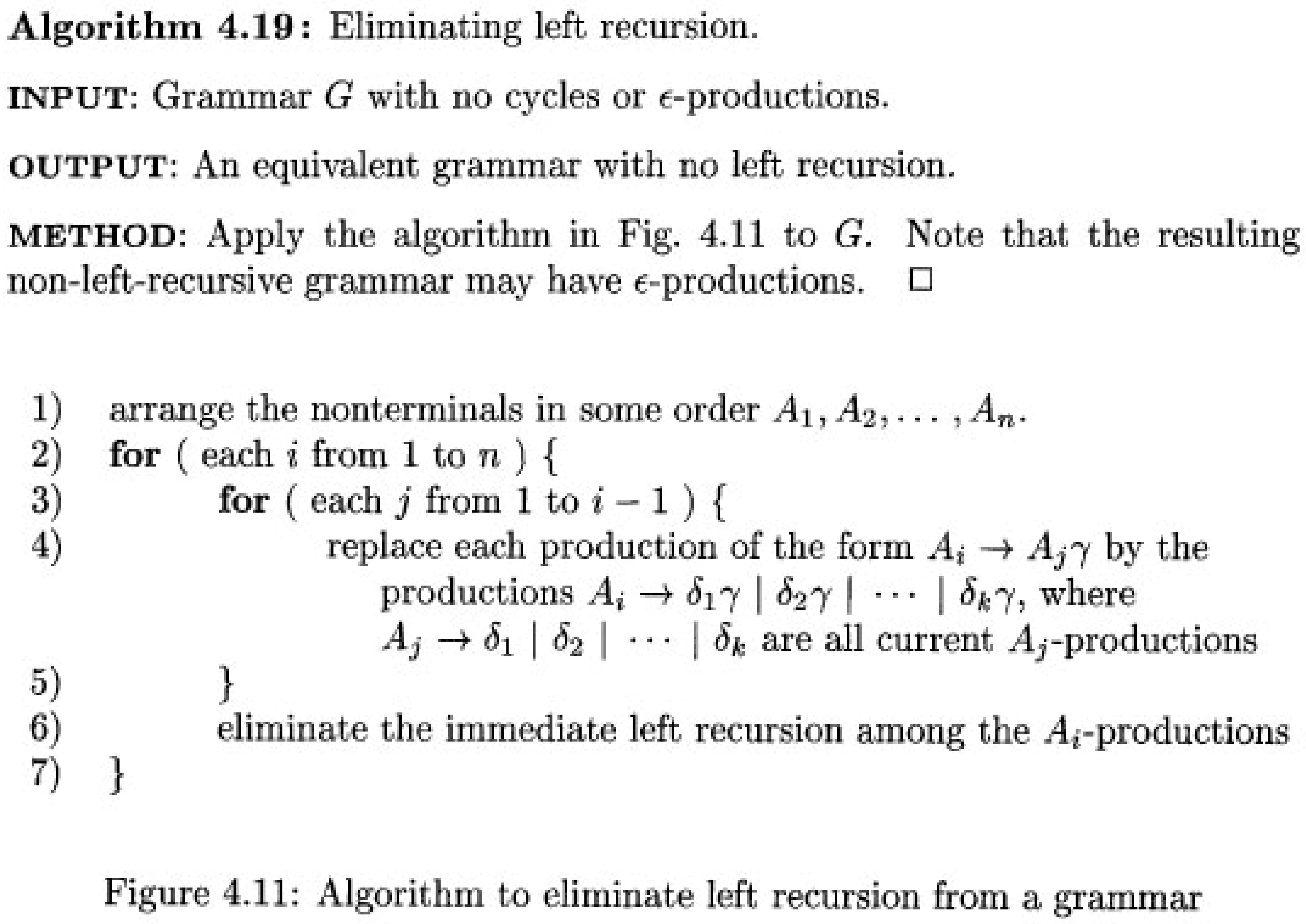
Elimination of Left Recursion (i)

- Top-down parsing methods cannot handle left recursion.
- We've already seen how to remove LR in previous lectures...problem is we've only looked at *immediate* LR (A->Aa).
- In the next slides we shall look at the general algorithm to remove left recursion … (A $-^{+}>$ Aa)

Elimination of Left Recursion (ii)

- $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
- Changes to
- $A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
- $A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid e$
- However check this grammar ….
  - S -> Aa | b
  - A -> Ac | Sd | ε

Elimination of Left Recursion (iii)



Algorithm 4.19 : Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
1)  arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2)  for ( each $i$ from 1 to $n$ ) {
3)          for ( each $j$ from 1 to $i-1$ ) {
4)                  replace each production of the form $A_i \to A_j\gamma$ by the
                        productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where
                        $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current $A_j$-productions
5)          }
6)          eliminate the immediate left recursion among the $A_i$-productions
7)  }
```

Figure 4.11: Algorithm to eliminate left recursion from a grammar

Left Factoring (i)

- Grammar transformation useful for predictive or top-down parsing.
- The idea is to delay the decision of which production to use until enough of the input is seen so that we can make the correct choice.
- *stmt*
  - **if** *expr* **then** *stmt* **else** *stmt*
  - **if** *expr* **then** *stmt*

Left Factoring (ii)

- In general, if we have

  A

  - $\alpha\beta_1 \mid \alpha\beta_2$
- We change this to

  A

  - $\alpha A'$

  A'

  - $\beta_1 \mid \beta_2$

Top-Down Parsing (i)

- Start from the root and create nodes for the parse tree in pre-order (depth-first)
- Finds a leftmost derivation for an input string
- At each step of a top-down parse
  - Determine the production to be applied for a non-terminal
  - And try to match terminal symbols in the production body with the input string
- We have already seen Predictive parsing, which is a special case of recursive-descent parsing.

Top-Down Parsing (ii)

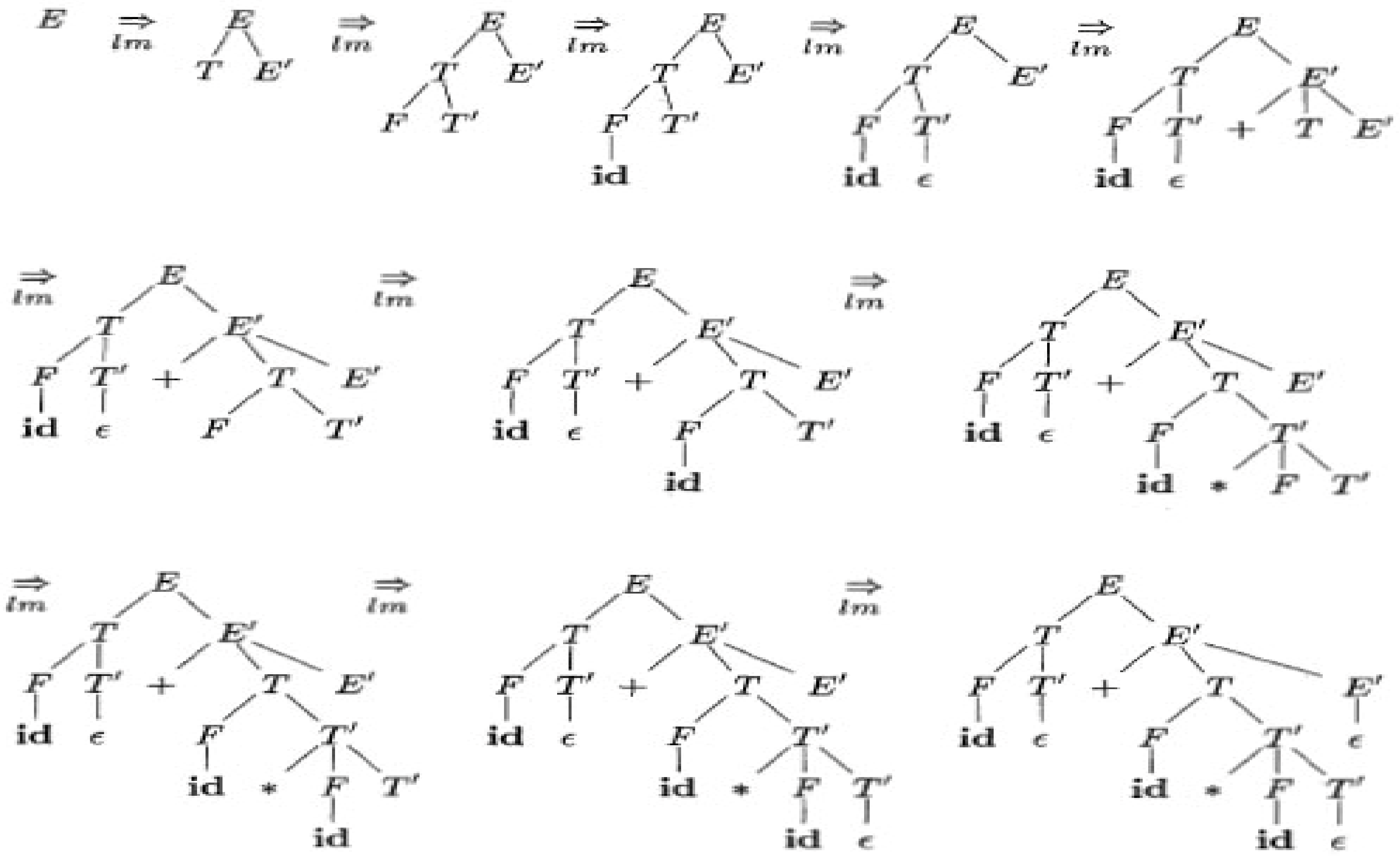
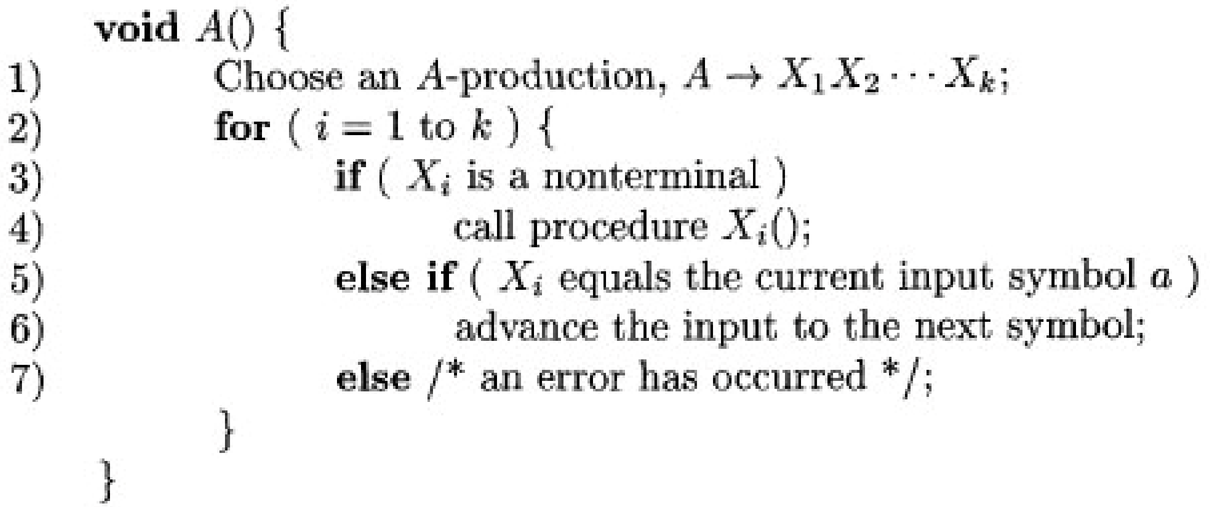


Figure 4.12: Top-down parse for **id** + **id** * **id**

Recursive-Descent Parsing Program

- Consists of a set of procedures, one for each non-terminal. In general it may require backtracking (which is not included in the code below). A Left Recursive grammar may cause a recursive-descent parser to go into an infinite loop.



```
void $A()$ {
        Choose an $A$-production, $A \rightarrow X_1X_2\cdots X_k$;
        for ( $i = 1$ to $k$ ) {
                if ( $X_i$ is a nonterminal )
                        call procedure $X_i()$;
                else if ( $X_i$ equals the current input symbol $a$ )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

Figure 4.13: A typical procedure for a nonterminal in a top-down parser

FIRST {set} ... then FOLLOW

- FIRST and FOLLOW are two important functions (which return sets) which aid in the construction on both top-down and bottom-up parsers.
- They will help in determining which production rule to apply, based on the next input symbol.
- FOLLOW is also used in panic-mode error recovery to generate the synchronisation tokens

First let us define FIRST

- First(α), where α is any string of grammar symbols
  - Set of terminals that begin strings derived from α.
- If $\alpha \Rightarrow^* \varepsilon$, then ε is also in FIRST(α)
- Recall how in a predictive parser we require that for A -> α | β, then FIRST(α) is disjoint from FIRST(β)
- The main idea here is that if the next non-terminal is in FIRST(α) then the parser should follow the production rule A -> α other if the non-terminal is in FIRST(β) then it should follow the production rule A -> β

In general …. FIRST

- If X is a terminal, then FIRST(X) = {X}
- If X is a non-terminal and $X \rightarrow Y_1Y_2\ldots Y_k$ is a production for some k >=1, then place ***a*** in FIRST(X) if for some i, ***a*** is in FIRST(Y_i), and ε is in all of FIRST(Y_1),…,FIRST(Y_{i-1}).
- If $X \rightarrow \varepsilon$ is a production then add ε to FIRST(X).

- e.g if $Y \Rightarrow^* \varepsilon$, then we add FIRST($Y_2$) in FIRST(X)

FOLLOW definition

- FOLLOW(A), for non-terminal A
  - The set of terminals a that can appear immediately to the right of A in some sentential form, alternatively
  - The set of terminals a such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$

- Note that in between A and a (above) there could be other non-terminals which can derive ε and disappear

compute .... FOLLOW

- Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right end marker
- If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is in FOLLOW(B)
- If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Examples of FIRST and FOLLOW

- Pg 222 of Aho contains various examples for computed FIRST and FOLLOW sets ... make sure you go through them.

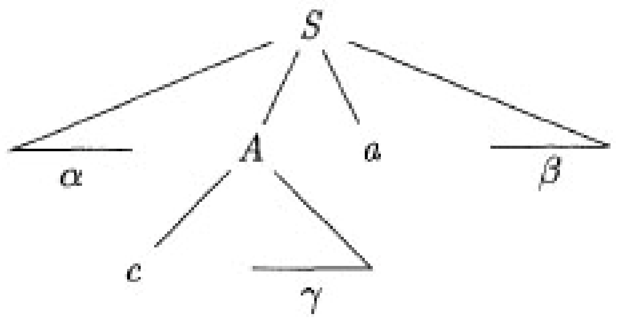


Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

LL(1) Grammars (i)

- First L stands for Left to Right scan of input
- Second L stands for Left-most derivation
- 1 stands for 1 symbol lookahead at each step to make parsing decisions.

- Can be parsed with a predictive parser (i.e. a recursive descent parser with no backtracking)
- Make sure that grammar is not left-recursive or ambiguous. These cannot be LL(1) grammars.

LL(1) Grammars Formally (ii)

- A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two disjoint productions of G, the following conditions hold:
  - For no terminal a do both α and β derive strings beginning with a,
  - At most one of α and β can derive ε,

An LL(1) grammar for statements

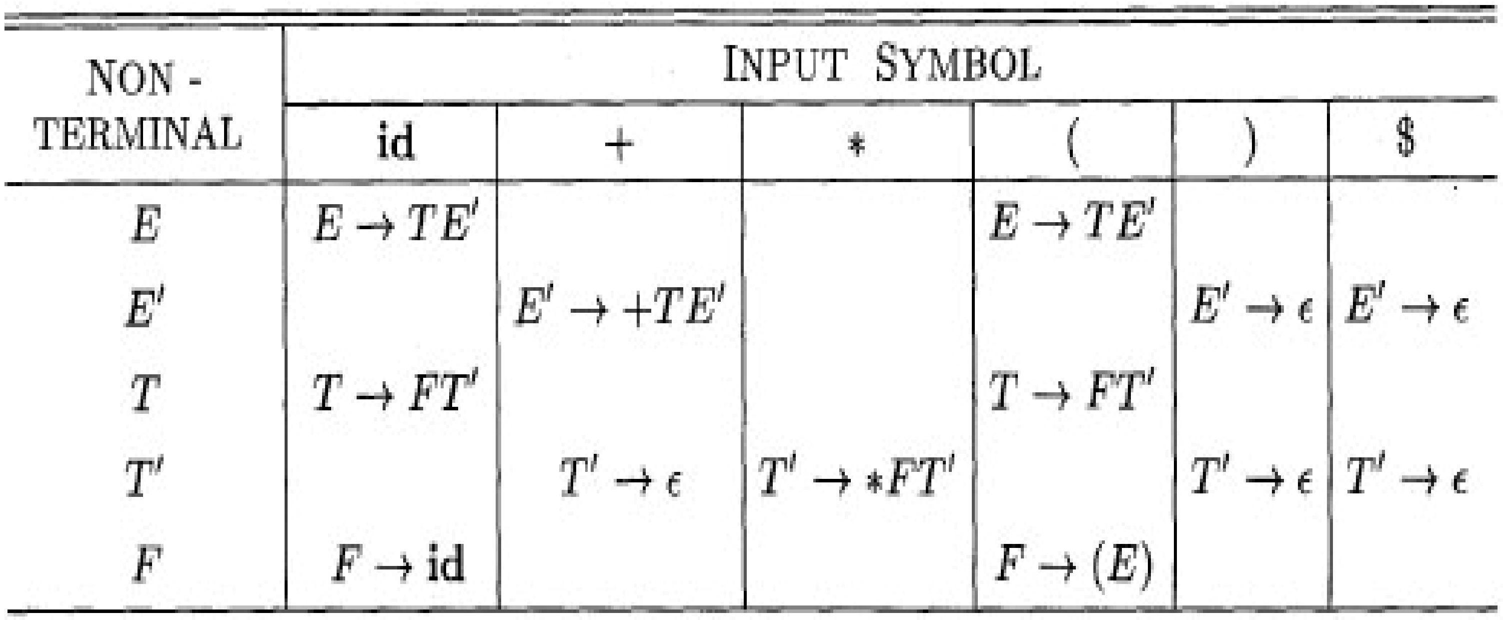
- Stmt -> if (expr) stmt else stmt
- Stmt -> while (expr) stmt
- Stmt -> { stmt_list }

Parsing Table - Predictive

- Non-terminals across y-axis and terminal symbols (+ $) across the X-axis.
- Construction Algorithm
  - For each production A -> α, do
    - For each terminal a in FIRST(A), add A -> α to M[A,a]
    - If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]. If ε is in FIRST(α) and $ is in FOLLOW(A), add A -> α to M[A,$] as well.

- The remaining empty cells indicate an error state !!

Parsing Table for LL(1) grammar in slide 5



NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Figure 4.17: Parsing table M for Example 4.32

Parsing Table (some entries)

- For production E -> T E'
  - FIRST(TE') = FIRST(T) = {(,id}
  - Production is added to M[E,(] and M[E,id]
- For production E' -> + T E'
  - FIRST(+TE') = {+}
  - Production is added to M[E',+]

Bottom-Up Parsing (i)

- Constructs a parse tree for an input string beginning at the leaves and working up towards the root.

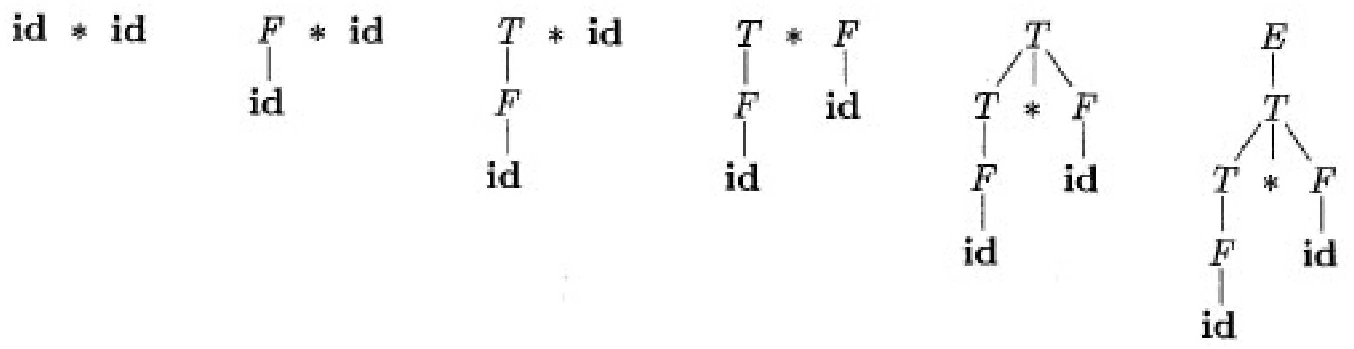


Figure 4.25: A bottom-up parse for **id** * **id**

Bottom-Up Parsing (ii)

- Bottom-up parsing is the process of *reducing* a string w to the start symbol of the grammar. Derivation in reverse !!

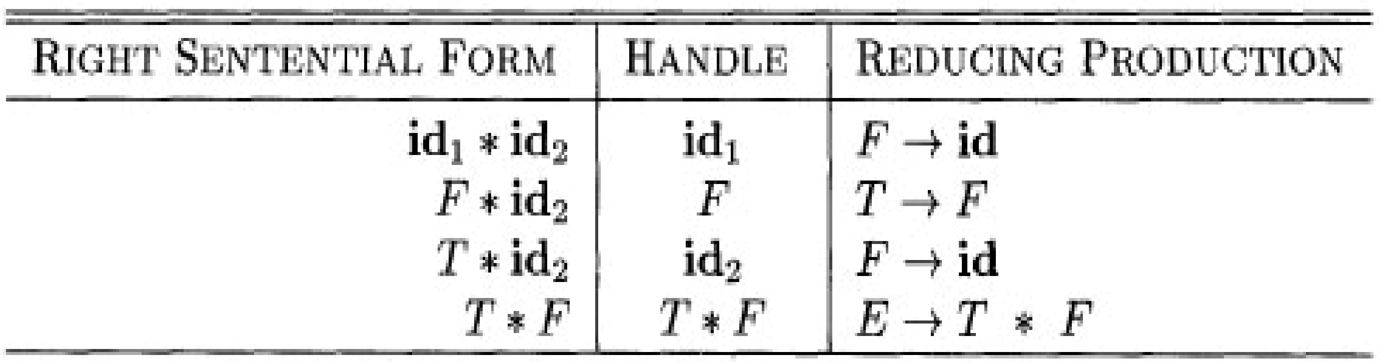
- At each *reduction* step, a specific substring matching the body of a production is replaced by the non terminal at the head of that production.

- The parser needs to decide when to reduce and what production to apply.

- id*id $\rightarrow$ F*id $\rightarrow$ T*id $\rightarrow$ T*F $\rightarrow$ T $\rightarrow$ E

Bottom-Up Parsing Handles

- Informally, a handle is a sub-string that matches the body of a production
- Its reduction represents one step along the reverse of a rightmost derivation.



RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

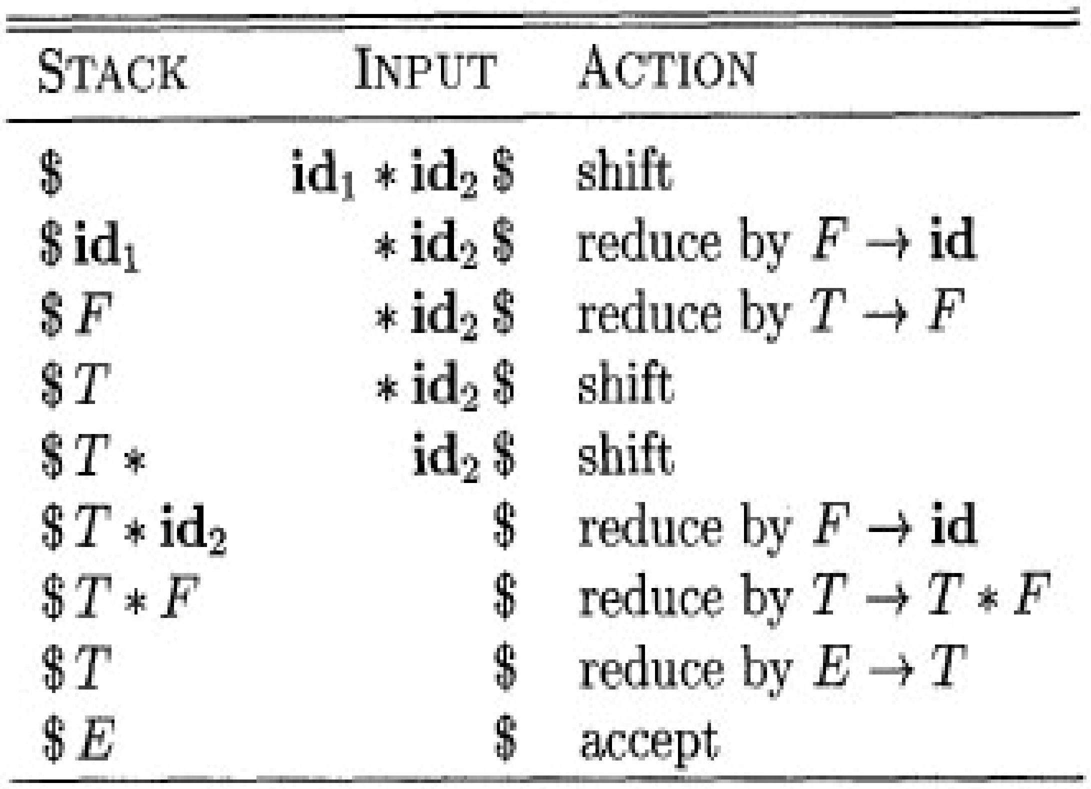
Shift-Reduce Parsing (i)

- Uses a stack to hold grammar symbols and an input buffer to hold the rest of the string to be parsed.
- We'll see that the handle will always appear at the top of the stack just before it is identified as a handle.
- $ is used to mark the bottom of the stack

Shift-Reduce Parsing (ii)

- During a left to right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce.
- This continues until either an error is discovered or when the top of the stack contains the start symbol.
- Important : we use a stack because the handle will always appear on top of it … never inside!

Shift-Reduce Parsing (iii)



STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\, \$$	shift
$\$\, \mathbf{id}_1$	$* \mathbf{id}_2\, \$$	reduce by $F \rightarrow \mathbf{id}$
$\$\, F$	$* \mathbf{id}_2\, \$$	reduce by $T \rightarrow F$
$\$\, T$	$* \mathbf{id}_2\, \$$	shift
$\$\, T *$	$\mathbf{id}_2\, \$$	shift
$\$\, T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\, T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$\, T$	$\$$	reduce by $E \rightarrow T$
$\$\, E$	$\$$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Shift-Reduce Parsing Operations

- **Shift** : shift the next input symbol onto the top of the stack
- **Reduce** : The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what non-terminal to replace the string
- **Accept** : Announce successful completion of parsing
- **Error** : Discover a syntax error and call an error recovery routine

Conflict During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsing cannot be used.
- Shift/Reduce conflict
  - Parser cannot decide whether to shift or to reduce
- Reduce/Reduce conflict
  - Parser cannot decide which rule to reduce

LR(k) Parsing

- "L" is for left-to-right scanning of the input
- "R" is for constructing a rightmost derivation in reverse
- "k" stands for the number of input symbols of lookahead that are used in making parsing decisions. For practical interest we have k=0 or 1.
- Efficient parser generators exist for LR grammars. (for eg YACC but not JavaCC which is LL)

Summary

- Top-down parsing
- Bottom up parsing
- Parser generators (e.g. JavaCC generates LL(k) parsers)