

## CHAPTER THREE

### 3. SYNTAX ANALYSIS

By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. For example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on. The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation (both are discussed in the previous course: formal language theory).

The use of CFGs has several advantages over BNF:

- helps in identifying ambiguities
- a grammar gives a precise yet easy to understand syntactic specification of a programming language
- it is possible to have a tool which produces automatically a parser using the grammar
- a properly designed grammar helps in modifying the parser easily when the language changes

#### 3.1. The Role of the Parser

Syntax analysis is also called parsing or hierarchical analysis. In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Figure 3.1, and verifies that the string of token names can be generated by the grammar for the source language. The grammar that a parser implements is called a Context Free Grammar or CFG. It is expected that the parser reports any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

The two parsers used in compilers are top-down and bottom-up. Top-down parser constructs the parse trees from top (root) to the bottom (leaves), while bottom-up parser builds the parse tree start from the leaves and work their way up to the root. Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing. In either case, the input to the parser is scanned from left to right, one symbol at a time.

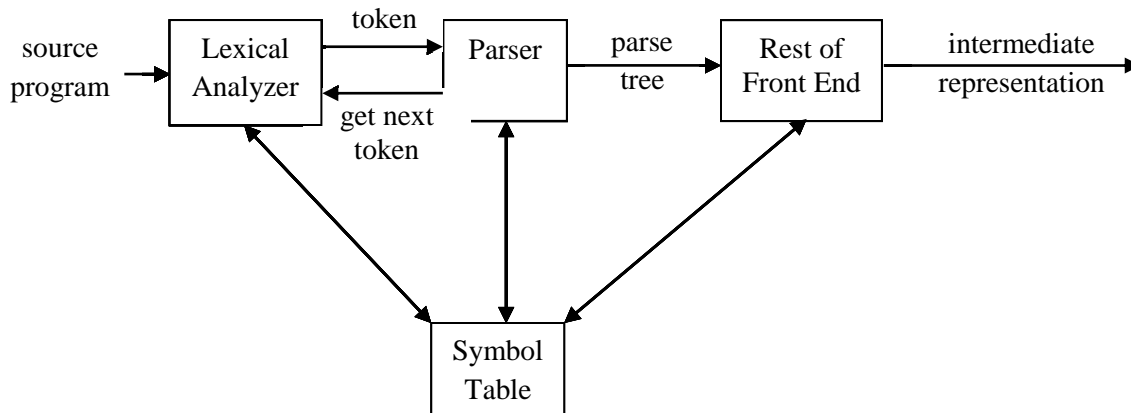


Figure 3.1: Position of parser in a compiler model

### 3.2. Syntax Error Handling

If a compiler had to process only correct programs, its design and implementation would be greatly simplified. But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.

Common programming errors can occur at many different levels.

- ✓ **Lexical errors** include misspellings of identifiers, keywords, or operators -e.g., the use of an identifier *ellipseSize* instead of *ellipseSize* - and missing quotes around text intended as a string.
- ✓ **Syntactic errors** include misplaced semicolons or extra or missing braces; that is, "{" or "}" . An arithmetic expression with unbalanced parentheses is syntactic error.
- ✓ **Semantic errors** include type mismatches between operators and operands. An operator applied to an incompatible operand is semantic error.
- ✓ **Logical errors** can be anything from incorrect reasoning on the part of the program. For example, the use of the assignment operator = instead of the comparison operator ==.

Often much of the error detection and recovery in a compiler is centered on the syntax analysis phase. One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyzer disobeys the grammatical rules defining the programming language. Another is the precision of modern parsing methods; they can detect the presence of syntactic errors in programs very efficiently. Accurately detecting the presence of semantic and logical errors at compile time is a much more difficult task.

The error handler in a parser has goals that are simple to state but challenging to realize:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

Fortunately, common errors are simple ones, and a relatively straightforward error-handling mechanism often suffices.

How should an error handler report the presence of an error? At the very least, it must report the place in the source program where an error is detected, because there is a good chance that the actual error occurred within the previous few tokens. A common strategy is to print the offending line with a pointer to the position at which an error is detected. The error handler also expected to report the type of error (if possible) for the detected errors.

Once an error is detected, how should the parser recover? Although no strategy has proven itself universally acceptable, a few methods have broad applicability.

There are **four main error recovery strategies in error handling**:

- **Panic-Mode Recovery**: with this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found (usually delimiters, such as }, whose role in the source program is clear and

unambiguous). While panic-mode correction often skips considerable amount of input without checking it for additional errors, it has the advantage of simplicity and is guaranteed not to go into an infinite loop

- **Phrase-Level Recovery**: On discovering an error, a parser may perform local correction on the remaining input such that the parser is able to continue. A typical local correction is to replace a comma by a semicolon, insert a missing semicolon, etc. We should be careful to avoid not going into infinite loop. Phrase-level replacement has been used in several error-reporting compilers. Its major **drawback** is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection
- **Error Productions**: By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. A parser constructed from a grammar augmented by these error productions detects the anticipated errors when the error production is used during parsing. The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input
- **Global Correction**: Ideally, we would like to make as few changes as possible in processing an incorrect input string. Given an incorrect input string  $x$  and grammar  $G$ , the algorithms will find a parse tree for a related string  $y$ , such that the number of insertions, deletions, and changes of tokens required to transform  $x$  into  $y$  is as small as possible. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

### 3.3. Context-Free Grammars

Context free grammar (shortly named as grammar) describes sets of strings (i.e. languages) and defines structure on these strings in the language it defines. It specifies allowable syntactic elements & rules for composing them from other syntactic elements, and also a set of rules to identify non-terminals to obtain from other terminals & non-terminals. Context free grammar is useful in specifying the syntactical structures of a programming language.

There are four components for a context free grammar, namely terminal, non-terminal, start and production rule.

1. *Terminals* are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer.
2. *Non-Terminals*: these components, also called syntactic variables, represent a set of strings of terminals, and they must be replaced by other things in a given grammar. Non-terminals are used to specify strings.
3. *Start symbol*: from non-terminals of a grammar has, one is designated as start symbol that specify the language, and the rest are used for specifying the string. Conventionally, the productions for the start symbol are listed first.

4. *Production rules*: the productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of:
  - a) A non-terminal called the *head* or *left* side of the production; this production defines some of the strings denoted by the head.
  - b) The symbol  $\rightarrow$
  - c) A *body* or *right* side consisting of zero or more terminals and non-terminals. The components of the body describe one way in which strings of the non-terminal at the head can be constructed.

For conventional notations, the following symbols are **terminals**:

- ✓ Lowercase letters early in the alphabet, such as  $a, b, c$
- ✓ Operator symbols such as  $+, *,$  and so on
- ✓ Punctuation symbols such as parentheses, comma, and so on
- ✓ The digits  $0, 1, \dots, 9$
- ✓ Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

The following symbols are **nonterminals**:

- ✓ Uppercase letters like  $A, B, C$ .
- ✓ The letter **S** is usually the start symbol.
- ✓ Unless stated otherwise, the head of the first production is the start symbol.

A set of productions  $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$  with a common head  $A$  (call them  $A$ -productions), may be written  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$ . Call  $\alpha_1, \alpha_2, \dots, \alpha_k$  the alternatives for  $A$ .

For example, take the following grammar and input string.

$E \rightarrow E - E \mid E * E \mid a \mid b \mid c$ ; and input string given is “ $a - b * c$ ”.

The production rule for the above grammar will be the following

$$\begin{aligned} E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow a \\ E &\rightarrow b \\ E &\rightarrow c \end{aligned}$$

For this grammar, the start symbol is  $E$ ; terminals are  $-, *, a, b, c$ ; and non-terminal is  $E$ . Left side of the arrow is head of the production, and right side of the arrow is body of the production.

### 3.3.1. Derivations

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

Derivation is a sequence of replacements of non-terminal symbols of the input string from the production rule.

For a general definition of derivation, consider a nonterminal  $A$  in the middle of a sequence of grammar symbols, as in  $\alpha A \beta$ , where  $\alpha$  and  $\beta$  are arbitrary strings of grammar symbols. Suppose  $A \rightarrow \gamma$  is a production. Then, we write  $\alpha A \beta \rightarrow \alpha \gamma \beta$ .

To decide which non-terminal to be replaced with production rule, we can have two options.

### ❖ Left - most Derivation (LMD)

If the sentential form of an input is scanned and replaced from left to right, it is called left most derivation. If we always choose the left-most non-terminal in each derivation step, this derivation is called as left-most derivation. It is simple one in which we replace the leftmost variable in a production body by one of its production bodies first, and then work our way from left to right.

Consider the input string **id + id \* id**, and productions

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow \text{id}$$

Then, the left most derivation will be the following.

$$E \rightarrow \underline{E} * E$$

$$E \rightarrow \underline{E} + E * E$$

$$E \rightarrow \text{id} + \underline{E} * E$$

$$E \rightarrow \text{id} + \text{id} * \underline{E}$$

$$E \rightarrow \text{id} + \text{id} * \text{id}$$

Notice that the left-most side non-terminal (underlined one) is always processed first.

### ❖ Right - most Derivation (RMD)

If we scan and replace the input with production rules, from right to left, it is known as rightmost derivation. If we always choose the right-most non-terminal in each derivation step, this derivation is called as right-most derivation. It is one in which we replace the rightmost variable by one of its production bodies first, and then work our way from right to left.

The right most derivation for the above example will be the following.

$$E \rightarrow E + \underline{E}$$

$$E \rightarrow E + E * \underline{E}$$

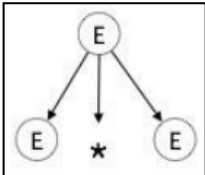
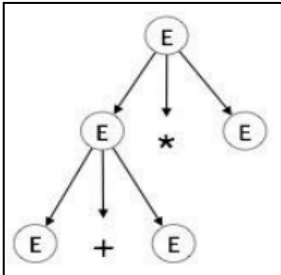
$$E \rightarrow E + \underline{E} * \text{id}$$

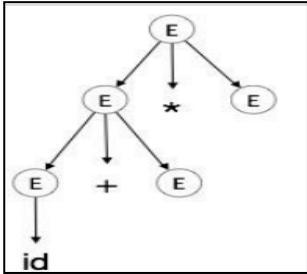
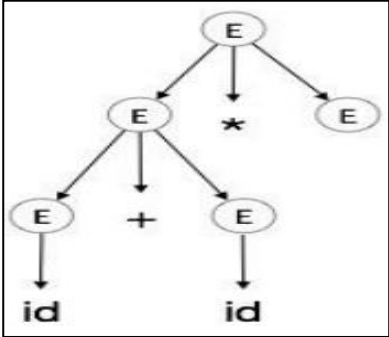
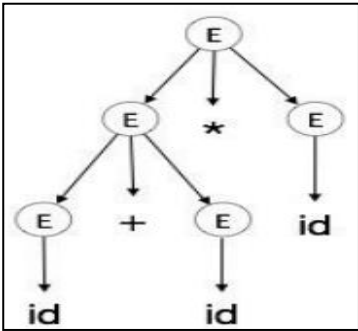
$$E \rightarrow \underline{E} + \text{id} * \text{id}$$

$$E \rightarrow \text{id} + \text{id} * \text{id}$$

### 3.3.2. Derivation and parse tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us take the above left most derivation and see the parse tree.

<b>Step 1:</b>	$E \rightarrow E * E$ 
<b>Step 2:</b>	$E \rightarrow E + E * E$ 

<b>Step 3:</b>	$E \rightarrow id + E * E$ 
<b>Step 4:</b>	$E \rightarrow id + id * E$ 
<b>Step 5:</b>	$E \rightarrow id + id * id$ 

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order.

Thus, a parse tree pictorially shows how the start symbol of a grammar derives a string in the language. Each interior node of a parse tree is labeled by some nonterminal  $A$ , and that the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this  $A$  was replaced in the derivation.

In a parse tree:

- ✓ The root of the tree is the start symbol
- ✓ All leaf nodes are terminals
- ✓ All interior nodes are non-terminals
- ✓ In-order traversal gives original input string

### 3.3.3. Ambiguity

A grammar  $G$  is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

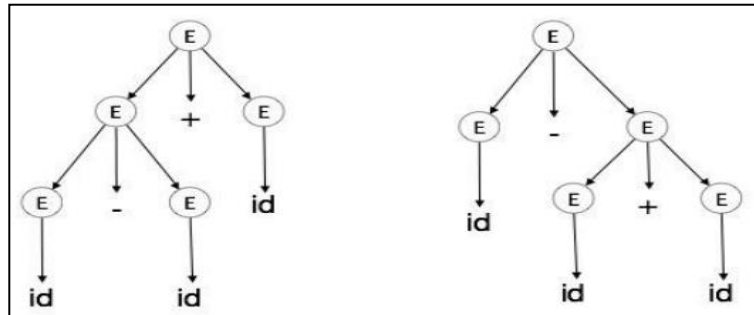
For example, consider the input string **id+id\*id** and the production.

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

$$E \rightarrow id$$

This grammar generates these two parse trees:



### 3.3.4. Context-Free Grammars Versus Regular Expressions

Grammars are a more powerful notation than regular expressions. Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa. Alternatively, every regular language is a context-free language, but not vice-versa.

For example, the regular expression **(a|b)\*abb** and the grammar

$$A_0 \rightarrow a A_0 \mid b A_0 \mid A_1$$

$$A_1 \rightarrow b A_2$$

$$A_2 \rightarrow b A_3$$

$$A_3 \rightarrow \varepsilon$$

describe the same language, the set of strings of  $a$ 's and  $b$ 's ending in  $abb$ .

On the other hand, the language  $L = \{a^n b^n \mid n \geq 1\}$  with an equal number of  $a$ 's and  $b$ 's is a prototypical example of a language that can be described by a grammar but not by a regular expression.

Regular expressions generally provide a more concise and easier to understand notation for tokens than grammars. In addition, more efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

Regular expressions are most useful for describing the structure of lexical constructs such as - identifiers, constants, keywords, and so forth. Grammars, on the other hand, are most useful in describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on.

## 3.4. Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first), which means that the top of a subtree is constructed before any of its lower nodes are. Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

At each step of top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say  $A$ . Once a production is chosen, the rest of parsing consists of "matching" the terminal symbols in the production body with the input string.

### 3.4.1. Recursive-Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and nonterminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require backtracking, that is, making repeated scans of the input. However, backtracking parsers are not seen frequently. One reason is that backtracking is rarely needed to parse programming language constructs.

To construct the parse tree using recursive-descent parsing:

- ✓ We create one node tree consisting of  $S$ .
- ✓ Two pointers, one for the tree and one for the input, will be used to indicate where the parsing process is. Initially, they will be on  $S$  and the first input symbol, respectively.
- ✓ Then we use the first  $S$ -production to expand the tree. The tree pointer will be positioned on the leftmost symbol of the newly created sub-tree.
- ✓ As the symbol pointed by the tree pointer matches that of the symbol pointed by the input pointer, both pointers are moved to the right.
- ✓ Whenever the tree pointer points on a non-terminal, we expand it using the first production of the non-terminal.
- ✓ Whenever the pointers point on different terminals, the production that was used is not correct, thus another production should be used. We have to go back to the step just before we replaced the non-terminal and use another production.
- ✓ If we reach the end of the input and the tree pointer passes the last symbol of the tree, we have finished parsing.

Example: Draw a parse tree for the input string  $cad$  using the following grammar

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

To construct a parse tree for the input string  $w = cad$ , begin with a tree consisting of a single node labeled  $S$ , and the input pointer pointing to  $c$ , the first symbol of  $w$ .  $S$  has only one production, so we use it to expand  $S$  and obtain the tree of Fig. 3.2 (a). The leftmost leaf, labeled  $c$ , matches the first symbol of input  $w$ , so we advance the input pointer to  $a$ , the second symbol of  $w$ , and consider the next leaf, labeled  $A$ .

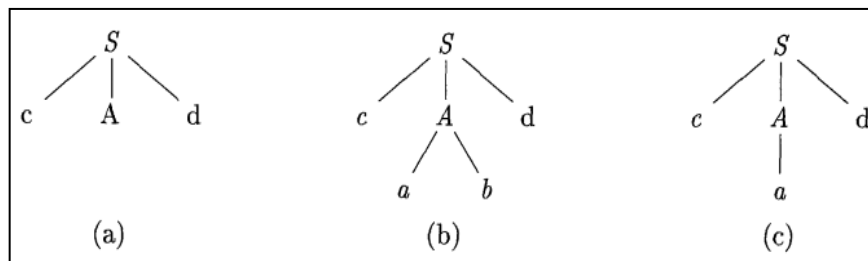


Figure 3.2: Steps in a top-down parse

Now, we expand  $A$  using the first alternative  $A \rightarrow ab$  to obtain the tree of Fig. 3.2 (b). We have a match for the second input symbol,  $a$ , so we advance the input pointer to  $d$ , the third input symbol, and compare  $d$  against the next leaf, labeled  $b$ . Since  $b$  does not match  $d$ , we report failure and go back to  $A$  to see whether there is another alternative for  $A$  that has not been tried, but that might produce a match.



In going back to  $A$ , we must reset the input pointer to position 2, the position it had when we first came to  $A$ , which means that the procedure for  $A$  must store the input pointer in a local variable.

The second alternative for  $A$  produces the tree of Fig. 3.2 (c). The leaf  $a$  matches the second symbol of  $w$  and the leaf  $d$  matches the third symbol. Since we have produced a parse tree for  $w$ , we halt and announce successful completion of parsing.

### 3.4.2. Predictive Parsing

A predictive parser also called non-Recursive Predictive Parser (or) Table Driven Predictive Parser is a top-down parsing that can be built by maintaining a stack explicitly, rather implicitly via recursive calls.

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(1) grammar.

This method, shown in Figure 3.3, uses a parsing table that determines the next production to be applied. It needs the following components to check whether the given string is successfully parsed or not.

- The input buffer contains the string to be parsed followed by  $\$$ . This is used to indicate that the input string is terminated. This is used as right end marker.
- The stack contains a sequence of grammar symbols. The grammar symbols will be either non-terminals or terminals. Initially the stack is pushed with “ $\$$ ” on the top of the stack. After that, as parsing progresses the grammar symbols are pushed. This “ $\$$ ” is used to announce the completion of parsing.
- The parsing table is generally a two dimensional array. An entry in the table is referred  $T[A, a]$ , where ‘ $A$ ’ is a non-terminal, ‘ $a$ ’ is a terminal or the symbol ‘ $\$$ ’ and ‘ $T$ ’ is a Table name.
- The parsing routine (output) works in conjunction with the parsing table. The output is a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer. This is a program which can be written in languages like C, C++ and Java.

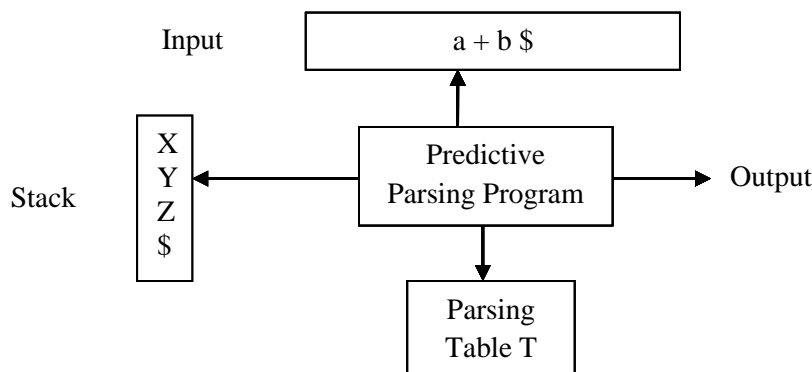


Figure 3.3: Model of a table-driven predictive parser

The parsing program reads from input buffer and considers the symbol on the top of the stack, and works in conjunction with parsing table  $T$ , and announces the output as successfully parsed or not.

### 3.4.3. First and Follow

The construction of both top-down and bottom-up parsers is aided by two functions, *FIRST* and *FOLLOW*, associated with a grammar  $G$ . During top-down parsing, *FIRST* and *FOLLOW* allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by *FOLLOW* can be used as synchronizing tokens.

$FIRST(\alpha)$ , for any string of grammar symbols  $\alpha$ , is the set of terminals that begin strings derived from  $\alpha$ . If  $\alpha \Rightarrow \epsilon$  in zero or more steps, then  $\epsilon$  is also in  $FIRST(\alpha)$ .

To compute  $FIRST(X)$  for all grammar symbols  $X$ , apply the following rules until no more terminals or  $\epsilon$  can be added to any *FIRST* set.

1. If  $X$  is a terminal, then  $FIRST(X) = \{X\}$
2. If  $X$  is a non-terminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production for some  $k \geq 1$ , then place  $\alpha$  in  $FIRST(X)$  if for some  $i$ ,  $\alpha$  is in  $FIRST(Y_i)$ , and  $\epsilon$  is in all of  $FIRST(Y_1), \dots, FIRST(Y_{i-1})$ ; that is  $Y_1 \dots Y_{i-1} \Rightarrow \epsilon$  in zero or more steps. If  $\epsilon$  is in  $FIRST(Y_j)$  for all  $j = 1, 2, \dots, k$ , then add  $\epsilon$  to  $FIRST(X)$ .
3. If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to  $FIRST(X)$ .

Now, we can compute *FIRST* for any string  $X_1 X_2 \dots X_n$  as follows

- ✓ Add to  $FIRST(X_1 X_2 \dots X_n)$  all non- $\epsilon$  symbols of  $FIRST(X_1)$ .
- ✓ Also add the non- $\epsilon$  symbols of  $FIRST(X_2)$  if  $\epsilon$  is in  $FIRST(X_1)$ ; the non- $\epsilon$  symbols of  $FIRST(X_3)$  if  $\epsilon$  is in  $FIRST(X_1)$  and  $FIRST(X_2)$ ; and so on
- ✓ Finally, add  $\epsilon$  to  $FIRST(X_1 X_2 \dots X_n)$  if, for all  $i$ ,  $\epsilon$  is in  $FIRST(X_i)$ .

$FOLLOW(A)$ , for a non-terminal  $A$ , is the set of terminals  $\alpha$  that can appear immediately to the right of  $A$  in some sentential form.

To compute  $FOLLOW(A)$  for all non-terminals  $A$ , apply the following rules until nothing can be added to any *FOLLOW* set.

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

For example, consider the following non-left-recursive grammar:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

1.  $FIRST(F) = FIRST(T) = FIRST(E) = \{ (, id \}$ . To see why, note that the two productions for  $F$  have bodies that start with these two terminal symbols, **id** and the left parenthesis  $($ .  $T$  has only one production, and its body starts with  $F$ . Since  $F$  does not derive  $\epsilon$ ,  $FIRST(T)$  must be the same as  $FIRST(F)$ . The same argument covers  $FIRST(E)$ .
2.  $FIRST(E') = \{ +, \epsilon \}$ . The reason is that one of the two productions for  $E'$  has a body that begins with terminal  $+$ , and the other's body is  $\epsilon$ . Whenever a nonterminal derives  $\epsilon$ , we place  $\epsilon$  in  $FIRST$  for that nonterminal.
3.  $FIRST(T') = \{ *, \epsilon \}$ . The reasoning is analogous to that for  $FIRST(E')$ .
4.  $FOLLOW(E) = FOLLOW(E') = \{ ), \$ \}$ . Since  $E$  is the start symbol,  $FOLLOW(E)$  must contain  $\$$ . The production body  $(E)$  explains why the right parenthesis  $)$  is in  $FOLLOW(E)$ . For  $E'$ , note that this non-terminal appears only **at the ends of bodies** of  $E$ -productions. Thus,  $FOLLOW(E')$  must be the same as  $FOLLOW(E)$ .
5.  $FOLLOW(T) = FOLLOW(T') = \{ +, ), \$ \}$ . Notice that  $T$  appears in bodies only followed by  $E'$ . Thus, everything except  $\epsilon$  that is in  $FIRST(E')$  must be in  $FOLLOW(T)$ ; that explains the symbol  $+$ . However, since  $FIRST(E')$  contains  $\epsilon$  (i.e.,  $E' \rightarrow \epsilon$ ), and  $E'$  is the entire string following  $T$  in the bodies of the  $E$ -productions, everything in  $FOLLOW(E)$  must also be in  $FOLLOW(T)$ . That explains the symbols  $\$$  and the right parenthesis  $)$ . As for  $T'$ , since it appears only at the ends of the  $T$ -productions, it must be that  $FOLLOW(T') = FOLLOW(T)$ .
6.  $FOLLOW(F) = \{ +, *, ), \$ \}$ . The reasoning is analogous to that for  $T$  in point (5).

#### 3.4.4. Construction of Predictive parsing table

Predictive parsing table is generally a two dimensional array. It has rows as non-terminals and columns as terminals, the symbol '\$' is used as the last column. The parsing routine works in conjunction with this table  $T$ .

The next algorithm collects the information from  $FIRST$  and  $FOLLOW$  sets into a predictive parsing table  $T[A, a]$ , a two-dimensional array, where  $A$  is a nonterminal, and  $a$  is a terminal or the symbol  $\$$ , the input end marker. The algorithm is based on the following idea: the production  $A \rightarrow a$  is chosen if the next input symbol  $a$  is in  $FIRST(a)$ . The only complication occurs when  $a = \epsilon$  or, more generally,  $a \rightarrow \epsilon$ . In this case, we should again choose  $A \rightarrow a$ , if the current input symbol is in  $FOLLOW(A)$ , or if the  $\$$  on the input has been reached and  $\$$  is in  $FOLLOW(A)$ .

**Algorithm:** Construction of a predictive parsing table.

**INPUT:** Grammar  $G$ .

**OUTPUT:** Parsing table  $T$ .

**METHOD :** For each production  $A \rightarrow X$  then do the following steps. ( $X$  is a grammar symbol).

1. For each terminal  $a$  in  $FIRST(X)$ , add  $A \rightarrow X$  to  $T[A, a]$ .
2. If  $FIRST(X)$  has  $\epsilon$ , then for each terminal  $b$  in  $FOLLOW(A)$ , add  $A \rightarrow X$  to  $T[A, b]$ .
3. If  $FIRST(X)$  has  $\epsilon$  and  $\$$  is in  $FOLLOW(A)$ , then add  $A \rightarrow X$  to  $T[A, \$]$ .

Consider the previous grammar again, and the first and follow computations for it.

$E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \epsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid \epsilon$   
 $F \rightarrow (E) \mid id$

Non-terminals	First	Follow
F	{(, id}	{+, *, ), \$}
T	{(, id}	{+, ), \$}
E	{(, id}	{), \$}
E'	{+, ε}	{), \$}
T'	{*, ε}	{+, ), \$}

The predictive parsing table is the following.

Non-terminals	Input Symbols					
	id	+	*	(	)	\$
<b>E</b>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<b>E'</b>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<b>T</b>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<b>T'</b>		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<b>F</b>	$F \rightarrow id$			$F \rightarrow (E)$		

Consider production  $E \rightarrow TE'$ . Since  $FIRST(TE') = FIRST(T) = \{ (, id \}$  this production is added to  $T[E, (]$  and  $T[E, id]$ . Production  $E' \rightarrow +TE'$  is added to  $T[E', +]$  since  $FIRST(+TE') = \{ + \}$ . Since  $FOLLOW(E') = \{ ), \$ \}$ , production  $E' \rightarrow \epsilon$  is added to  $T[E', )]$  and  $T[E', \$]$ .

### 3.4.5. LL(1) Grammars

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1). The first "L" in LL (1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of look ahead at each step to make parsing action decisions.

The class of LL(1) grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language. For example, a left recursive grammar cannot be a LL(1) grammar. If a grammar is not left factored, it cannot be a LL(1) grammar. An ambiguous grammar cannot be a LL(1) grammar.

LL(1) grammars have several distinct properties. No ambiguous or left-recursive grammar can be LL(1). It can also be shown that a grammar  $G$  is LL(1) if and only if whenever  $A \rightarrow \alpha \mid \beta$  are two distinct productions of  $G$ , the following conditions hold:

1. For no terminal  $a$  do both  $\alpha$  and  $\beta$  derive strings beginning with  $a$ . That is, both  $\alpha$  and  $\beta$  cannot derive strings starting with same terminals.
2. At most one of  $\alpha$  and  $\beta$  can derive the empty string, i.e.  $\epsilon$
3. If  $\beta \Rightarrow \epsilon$  in zero or more steps, then  $\alpha$  does not derive any string beginning with a terminal in  $\text{FOLLOW}(A)$ . Likewise, if  $\alpha \Rightarrow \epsilon$  in zero or more steps, then  $\beta$  does not derive any string beginning with a terminal in  $\text{FOLLOW}(A)$ .

The first two conditions are equivalent to the statement that  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$  are disjoint sets. The third condition is equivalent to stating that if  $\epsilon$  is in  $\text{FIRST}(\beta)$ , then  $\text{FIRST}(\alpha)$  and  $\text{FOLLOW}(A)$  are disjoint sets, and likewise if  $\epsilon$  is in  $\text{FIRST}(\alpha)$ .

Note: LL(1) grammar is a grammar for which the parsing table does not have a multiply-defined entries

Example: Let a grammar  $G$  be given by:

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

Is the grammar  $G$  a LL(1) grammar?

Non-terminal	Input Symbols					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

From the above parsing table, the entry for  $T[S', e]$  contains both  $S' \rightarrow eS$  and  $S' \rightarrow \epsilon$ . The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an  $e$  is seen. Hence the grammar is not LL(1) grammar.

### 3.5. Bottom-Up Parsing

Bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). The bottom-up parsing method constructs the nodes in the parse tree in postorder: the top of a subtree is constructed after all of its lower nodes have been constructed. Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.

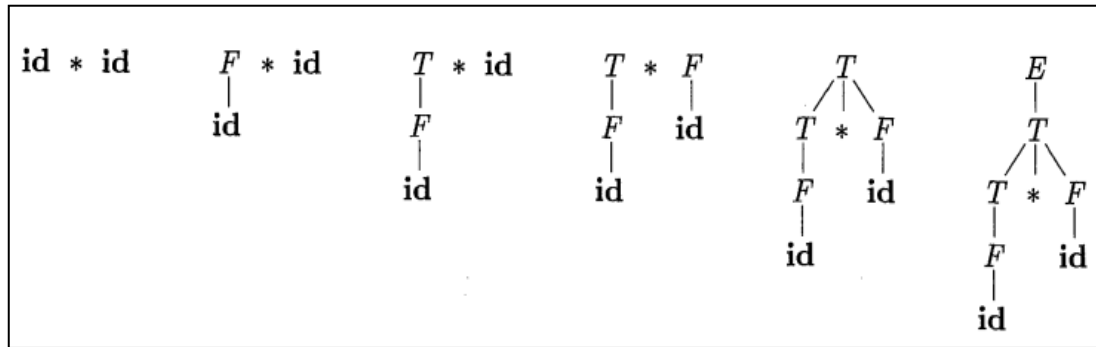
If the start symbol of the grammar can be obtained, from the input string, then the string is said to be accepted by the language. The input string is reduced by the given productions of the grammar so that the start symbol is obtained. Bottom-up parsers attempt to find the right-most derivation in reverse for given input string.

The sequence of tree snapshots in the following figure illustrates a bottom-up parse of the token stream **id \* id**, with respect to the expression of the following grammar.

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

Figure 3.4: A bottom-up parse for **id \* id**

Example: consider the grammar

$S \rightarrow aABe$

$A \rightarrow Abc|b$

$B \rightarrow d$

Given the input string **abbcd**e, which has to be checked for its acceptance by the language.

Using bottom-up parsing, the acceptance is as shown below.

**abbcd**e

**aAbcd**e [ $A \rightarrow b$ ]

**aAde** [ $A \rightarrow Abc$ ]

**aABe** [ $B \rightarrow d$ ]

**S** [ $S \rightarrow aABe$ ]

Thus the start symbol 'S' is obtained, which shows the acceptance of the string **abbcd**e by the grammar. This is the reverse of the right most derivation, where the rightmost derivation for the same is shown below,

$S \rightarrow aABe$

$\rightarrow aAde$

$\rightarrow aAbcde$

$\rightarrow abbcde$

- We will see a general style of bottom-up parsing called shift-reduce parsing.

### 3.5.1. Reductions

We can think of bottom-up parsing as the process of "reducing" a string  $w$  to the start symbol of the grammar. At each reduction step, a particular substring matching the right side of a production (body) is replaced by the symbol of the left of that production (head), and if the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

For example, the snapshots in figure 3.4 illustrate a sequence of reductions for the given grammar. The reductions will be discussed in terms of the sequence of strings as follows.

**id \* id** , **F \* id**, **T \* id**, **T \* F**, **T**, **E**

The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string **id\*id**. The first reduction produces **F \* id** by reducing the

leftmost **id** to **F**, using the production  $F \rightarrow \mathbf{id}$ . The second reduction produces  $T * \mathbf{id}$  by reducing **F** to **T**.

Now, we have a choice between reducing the string **T**, which is the body of  $E \rightarrow T$ , and the string consisting of the second **id**, which is the body of  $F \rightarrow \mathbf{id}$ . Rather than reduce **T** to **E**, the second **id** is reduced to **T**, resulting in the string  $T * F$ . This string then reduces to **T**. The parse completes with the reduction of **T** to the start symbol **E**.

By definition, a reduction is the reverse of a step in a derivation (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in Fig. 3.4:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

This derivation is in fact a rightmost derivation.

### 3.5.2. Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.

A shift-reduce parser tries to reduce the given input string into the starting symbol.

A string -----→ the starting symbol  
reduced to

We use **\$** to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty, and the string **w** is on the input as follows:

Stack	Input
\$	w\$

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 a string  $\beta$  of grammar symbols on top of the stack. It then reduces  $\beta$  to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

Stack	Input
\$E	\$

Upon entering this configuration, the parser halts and announces successful completion of parsing. The following table shows the steps through the actions a shift-reduce parser might take in parsing input string  $\mathbf{id}_1 * \mathbf{id}_2$  over the previous expression grammar.

Stack	Input	Action
\$	$\mathbf{id}_1 * \mathbf{id}_2$ \$	shift
\$ $\mathbf{id}_1$	$* \mathbf{id}_2$ \$	reduce by $F \rightarrow \mathbf{id}$
\$ <b>F</b>	$* \mathbf{id}_2$ \$	reduce by $T \rightarrow F$
\$ <b>T</b>	$* \mathbf{id}_2$ \$	shift
\$ <b>T*</b>	$\mathbf{id}_2$ \$	shift
\$ $\mathbf{T} * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $\mathbf{T} * \mathbf{F}$	\$	reduce by $T \rightarrow T * F$
\$ <b>T</b>	\$	reduce by $E \rightarrow T$
\$ <b>E</b>	\$	accept

Table: Configuration of a shift-reduce parser on input  $\mathbf{id}_1 * \mathbf{id}_2$

While the primary operations are shift and reduce, there are four possible actions a shift-reduce parser can make: shift, reduce, accept, and error.

1. *Shift*. Shift the next input symbol on the top of the stack.
2. *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 nonterminal to replace the string.
3. *Accept*. Announce successful completion of parsing.
4. *Error*. Discover a syntax error and call an error recovery routine.

### 3.6. Introduction to LR Parsing

The most prevalent type of bottom-up parsers today is based on a concept called LR(k) parsing; the “L” is for left-to-right scanning of the input, the “R” for constructing a rightmost derivation in reverse, and the k for the number of input symbols of look ahead that are used in making parsing decisions. The cases  $k = 0$  or  $k = 1$  are of practical interest, and we shall only consider LR parsers with  $k \leq 1$  here. When (k) is omitted, k is assumed to be 1.

This section introduces the basic concepts of LR parsing and the easiest method for constructing shift-reduce parsers, called “simple LR” (or SLR, for short).

There are two more complex methods – canonical-LR and LALR – that are used in the majority of LR parsers. [Read about these parsers].

#### The LR-Parsing Algorithm

A schematic representation of an LR parser is shown in Figure 3.5. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (action and goto). The driver program is the same for all LR parsers, only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time.

- The program uses a stack to store a string of the form  $s_0X_1s_1X_2 \dots X_ms_m$ , where  $s_m$  is on top
- Each  $X_i$  is a grammar symbol and each  $s_i$  is a symbol called a *state*
- Each state symbol summarizes the information contained in the stack below it, and the combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shift-reduce parsing decision

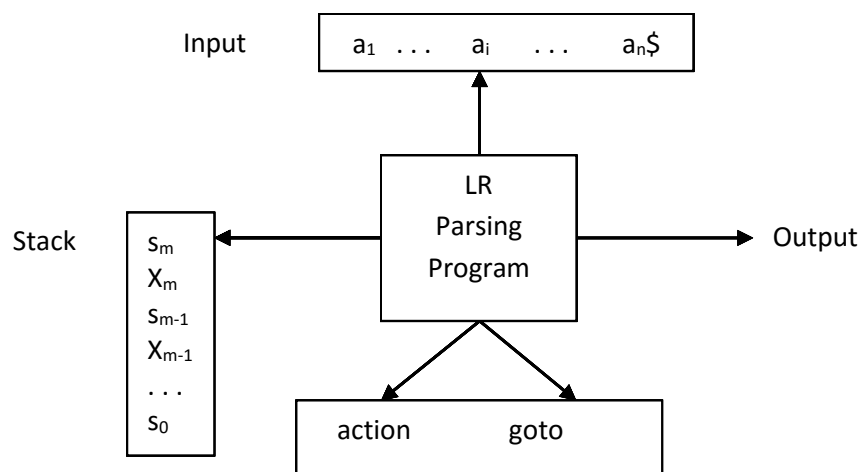


Figure 3.5: Model of an LR parser



The parsing table consists of two parts: a parsing-action function action and a goto function goto. The program driving the LR parser behaves as follows:

- It determines  $s_m$ , the state currently on top of the stack, and  $a_i$ , the current input symbol. It then consults  $\text{action}[s_m, a_i]$ , the parsing action table entry for state  $s_m$  and input  $a_i$ , which can have one of four values:

1. shift  $s$ , where  $s$  is a state,
2. reduce by a grammar production  $A \rightarrow \beta$ ,
3. accept, and
4. error

The function goto takes a state and grammar symbol as arguments and produces a state. A *configuration* of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpanded input:

$$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$$

This configuration represents a right-sentential form  $X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$  is essential the same way as a shift-reduce parser would; only the presence of states on the stack is new. The next move of the parser is determined by reading  $a_i$ , the current input symbol, and  $s_m$ , the state on top of the stack, and then consulting the parsing action table entry  $\text{action}[s_m, a_i]$

The configurations resulting after each of the four types of move are as follows:

1. If  $\text{action}[s_m, a_i] = \text{shift } s$ , the parser executes a shift move, entering the configuration  $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$ . Here the parser has shifted both the current input symbol  $a_i$  and the next state  $s$ , which is given in  $\text{action}[s_m, a_i]$ , onto the stack;  $a_{i+1}$  become the current input symbol.
2. If  $\text{action}[s_m, a_i] = \text{reduce } A \rightarrow \beta$ , then the parser executes a reduce move, entering the configuration  $(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$  where  $s = \text{goto}[s_{m-r}, A]$  and  $r$  is the length of  $\beta$ , the right side of the production. Here the parser first popped  $2r$  symbols off the stack ( $r$  state symbols and  $r$  grammar symbols), exposing state  $s_{m-r}$ . The parser then pushed both  $A$ , the left side of the production, and  $s$ , the entry for  $\text{goto}[s_{m-r}, A]$ , onto the stack. The current input symbol is not changed.
3. If  $\text{action}[s_m, a_i] = \text{accept}$ , parsing is completed
4. If  $\text{action}[s_m, a_i] = \text{error}$ , the parser has discovered an error and call an error recovery routine.

State	action						goto		
	id	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figure 3.6: Parsing table for expression grammar

1. si means shift and stack state i,
  2. rj means reduce by a production numbered j,
  3. acc means accept
  4. blank means error
- Figure 3.6 shows the parsing action and goto functions of an LR parsing table for arithmetic expressions with binary operators + and \*:
    1.  $E \rightarrow E+T$
    2.  $E \rightarrow T$
    3.  $T \rightarrow T*F$
    4.  $T \rightarrow F$
    5.  $F \rightarrow (E)$
    6.  $F \rightarrow id$

On input **id\*id+id**, the sequence of stack and input contents is shown in Figure 3.7

Line	Stack	Symbols	Input	Action
(1)	0		id*id+id\$	Shift to 5
(2)	05	id	*id+id\$	Reduce by $F \rightarrow id$
(3)	03	F	*id+id\$	Reduce by $T \rightarrow F$
(4)	02	T	*id+id\$	Shift to 7
(5)	027	T*	id+id\$	Shift to 5
(6)	0275	T*id	+id\$	Reduce by $F \rightarrow id$
(7)	02710	T*F	+id\$	Reduce by $T \rightarrow T*F$
(8)	02	T	+id\$	Reduce by $E \rightarrow T$
(9)	01	E	+id\$	Shift
(10)	016	E+	id\$	Shift
(11)	0165	E+id	\$	Reduce by $F \rightarrow id$
(12)	0163	E+F	\$	Reduce by $T \rightarrow F$
(13)	0169	E+T	\$	Reduce by $E \rightarrow E+T$
(14)	01	E	\$	accept

Figure 3.7 Moves of LR parser on **id\*id+id**

On input **id \* id + id**, the sequence of stack and input contents is shown in Fig. 4.37. Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack. For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with id the first input symbol. The action in row 0 and column id of the action field of Fig. 4.36 is s5, meaning shift by pushing state 5. That is what has happened at line (2): the state symbol 5 has been pushed onto the stack, and id has been removed from the input.

Then, \* becomes the current input symbol, and the action of state 5 on input \* is to reduce by  $F \rightarrow id$ . One state symbol is popped off the stack. State 0 is then exposed. Since the goto of state

0 on F is 3, state 3 is pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves is determined similarly.

### Constructing SLR Parsing Tables

This method is the simplest of the three methods used to construct an LR parsing table. It is called SLR (simple LR) because it is the easiest to implement. However, it is also the weakest in terms of the number of grammars for which it succeeds. A parsing table constructed by this method is called SLR table. A grammar for which an SLR table can be constructed is said to be an SLR grammar.

The SLR method for constructing parsing tables is a good starting point for studying LR parsing. We shall refer to the parsing table constructed by this method as an SLR table, and to an LR parser using an SLR-parsing table as an SLR parser. The other two methods augment the SLR method with look ahead information.

The SLR method begins with LR(0) items and LR(0) automata. That is, given a grammar,  $G$ , we augment  $G$  to produce  $G'$ , with a new start symbol  $S'$ . From  $G'$ , we construct  $C$ , the canonical collection of sets of items for  $G'$  together with the GOTO function.

The ACTION and GOTO entries in the parsing table are then constructed using the following algorithm. It requires us to know FOLLOW( $A$ ) for each nonterminal  $A$  of a grammar.

**Algorithm:** Constructing an SLR-parsing table.

**Input:** An augmented grammar  $G'$ .

**Output:** The SLR-parsing table functions ACTION and GOTO for  $G'$ .

**Method:**

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of sets of LR(0) items for  $G'$ .
2. State  $i$  is constructed from  $I_i$ . The parsing actions for state  $i$  are determined as follows:
  - (a) If  $[A \rightarrow \alpha.a\beta]$  is in  $I_i$ , and  $GOTO(I_i, a) = I_j$ , then set ACTION[ $i, a$ ] to "shift  $j$ ." Here  $a$  must be a terminal.
  - (b) If  $[A \rightarrow \alpha.]$  is in  $I_i$ , then set ACTION[ $i, a$ ] to "reduce  $A \rightarrow \alpha$ " for all  $a$  in FOLLOW( $A$ ); here  $A$  may not be  $S'$ .
  - (c) If  $[S' \rightarrow S.]$  is in  $I_i$ , then set ACTION[ $i, \$$ ] to "accept."

If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state  $i$  are constructed for all nonterminals  $A$  using the rule: If  $GOTO(I_i, A) = I_j$ , then  $GOTO[i, A] = j$ .
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow .S]$ .

*Reading assignment: Read how SLR parsing table is constructed.*