



**SATHYABAMA**

INSTITUTE OF SCIENCE AND TECHNOLOGY  
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

[www.sathyabama.ac.in](http://www.sathyabama.ac.in)

**SCHOOL OF COMPUTING**  
**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**UNIT – II - Compiler Design – SCS1303**

## II. PARSER

CFG – Derivation – CFG vs R.E. - Types of Parser –Bottom Up: Shift Reduce Parsing - Operator Precedence Parsing, SLR parser- Top Down: Recursive Decent Parser - Non-Recursive Decent Parser.

### SYNTAX ANALYSIS:

Every programming language has rules that prescribe the syntactic structure of well-formed programs. In Pascal, for example, a program is made out of blocks, a block out of statements, a statement out of expressions, an expression out of tokens, and so on. The syntax of programming language constructs can be described by context-free grammars or BNF (Backus-Naur Form) notation. Grammars offer significant advantages to both language designers and compiler writers.

- A grammar gives a precise, yet easy-to-understand. Syntactic specification of a programming language.
- From certain classes of grammars we can automatically construct an efficient parser that determines if a source program is syntactically well formed. As an additional benefit, the parser construction process can reveal syntactic ambiguities and other difficult-to-parse constructs that might otherwise go undetected in the initial design phase of a language and its compiler.
- A properly designed grammar imparts a structure to a programming language that is useful for the translation of source programs into correct object code and for the detection of errors. Tools are available for converting grammar-based descriptions of translations into working pro-grams.

Languages evolve over a period of time, acquiring new constructs and performing additional tasks. These new constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the language.

### ROLE OF THE PARSER:

Parser for any grammar is program that takes as input string  $w$  (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for  $w$ , if  $w$  is a valid sentences of grammar or error message indicating that  $w$  is not a valid sentences of given grammar.

The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of

derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementary subtree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.

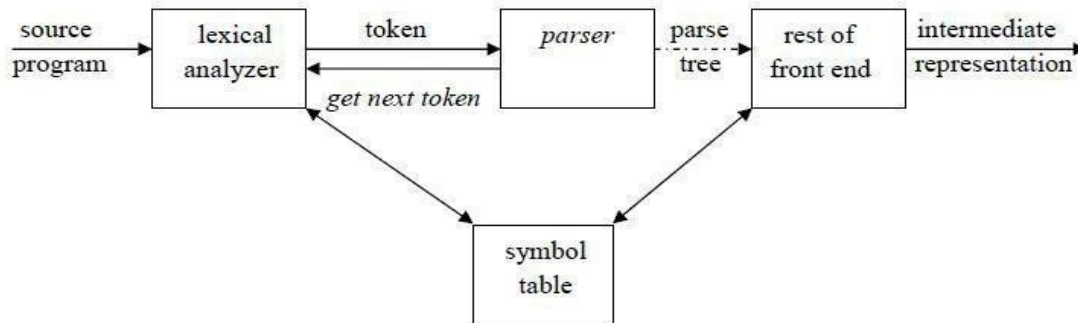


Fig 2.1 Role of Parser

## CONTEXT FREE GRAMMARS

A context-free grammar (grammar for short) consists of terminals, non-terminals, a start symbol, and productions.

1. Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about grammars for programming languages.
2. Non terminals are syntactic variables that denote sets of strings. They also impose a hierarchical structure on the language that is useful for both syntax analysis and translation.
3. In a grammar, one non terminal is distinguished as the start symbol, and the set of strings it denotes is the language defined by the grammar.
4. 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 non terminal, followed by an arrow, followed by a string of non terminals and terminals.

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples  $G(V, T, P, S)$ . Here,  $V$  is finite set of terminals (in our case, this will be the set of tokens)  $T$  is a finite set of non-terminals (syntactic- variables).  $P$  is a finite set of productions rules in the following form  $A \rightarrow \alpha$  where  $A$  is a non- terminal and  $\alpha$  is a string of terminals and non-terminals (including the empty string).  $S$  is a start symbol (one of the non-terminal symbol).

$L(G)$  is the language of  $G$  (the language generated by  $G$ ) which is a set of sentences.

A sentence of  $L(G)$  is a string of terminal symbols of  $G$ . If  $S$  is the start symbol of  $G$  then  $\omega$  is a sentence of  $L(G)$  iff  $S \Rightarrow \omega$  where  $\omega$  is a string of terminals of  $G$ . If  $G$  is a context-free grammar (CFG) then  $L(G)$  is a context-free language. Two grammars  $G_1$  and  $G_2$  are equivalent, if they produce same grammar.

Consider the production of the form  $S \Rightarrow \alpha$ , If  $\alpha$  contains non-terminals, it is called as a sentential form of  $G$ . If  $\alpha$  does not contain non-terminals, it is called as a sentence of  $G$ .

**Example:** Consider the grammar for simple arithmetic expressions:

$expr \rightarrow expr \ op \ expr$

$expr \rightarrow ( \ expr \ )$

$expr \rightarrow - \ expr$

$expr \rightarrow \mathbf{id}$

$op \rightarrow +$

$op \rightarrow -$

$op \rightarrow *$

$op \rightarrow /$

$op \rightarrow \wedge$

Terminals :  $id \ + \ - \ * \ / \ \wedge \ ( \ )$

Non-terminals :  $expr \ , \ op$

Start symbol :  $expr$

### Notational Conventions:

1. These symbols are terminals:
  - i. Lower-case letters early in the alphabet such as a, b, c.
  - ii. Operator symbols such as +, -, etc.
  - iii. Punctuation symbols such as parentheses, comma etc.
  - iv. Digits 0,1,...,9.
  - v. Boldface strings such as **id** or **if** (keywords)
2. These symbols are non-terminals:
  - i. Upper-case letters early in the alphabet such as A, B, C..
  - ii. The letter *S*, when it appears is usually the start symbol.
  - iii. Lower-case italic names such as *expr* or *stmt*.
3. Upper-case letters late in the alphabet, such as X,Y,Z, represent grammar symbols,

that is either terminals or non-terminals.

4. Greek letters  $\alpha$ ,  $\beta$ ,  $\gamma$  represent strings of grammar symbols.

e.g a generic production could be written as  $A \rightarrow \alpha$ .

5. If  $A \rightarrow \alpha_1$ ,  $A \rightarrow \alpha_2$ ,  $\dots$ ,  $A \rightarrow \alpha_n$  are all productions with  $A$ , then we can write  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ , (alternatives for  $A$ ).
6. Unless otherwise stated, the left side of the first production is the start symbol.

Using the shorthand, the grammar can be written as:

$$E \rightarrow E A E \mid ( E ) \mid - E \mid \mathbf{id}$$

$$A \rightarrow + \mid - \mid * \mid / \mid ^$$

### Derivations:

A *derivation* of a string for a grammar is a sequence of grammar rule applications that transform the start symbol into the string. A derivation proves that the string belongs to the grammar's language.

$\Rightarrow$  derives in one step

$\overset{+}{\Rightarrow}$  derives in  $\geq$  one step

$\overset{*}{\underset{G}{\Rightarrow}}$  indicates that the derivation utilizes the rules of grammar  $G$

### To create a string from a context-free grammar:

- Begin the string with a start symbol.
- Apply one of the production rules to the start symbol on the left-hand side by replacing the start symbol with the right-hand side of the production.
- Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right-hand side of some corresponding production, until all non-terminals have been replaced by terminal symbols.

In general a derivation step is  $\alpha A \beta \rightarrow \alpha \gamma \beta$  is sentential form and if there is a production rule  $A \rightarrow \gamma$  in our grammar. where  $\alpha$  and  $\beta$  are arbitrary strings of terminal and non-terminal symbols  $\alpha_1 \alpha_2 \dots \alpha_n$  ( $\alpha_n$  derives from  $\alpha_1$  or  $\alpha_1$  derives  $\alpha_n$ ). There are two types of derivation:

#### 1. Leftmost Derivation (LMD):

- If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.
- The sentential form derived by the left-most derivation is called the left-sentential form.

## 2. Rightmost Derivation (RMD):

- If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.
- The sentential form derived from the right-most derivation is called the right-sentential form.

### Example:

Consider the G,

$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \mathbf{id}$

Derive the string  $\mathbf{id + id * id}$  using leftmost derivation and rightmost derivation.

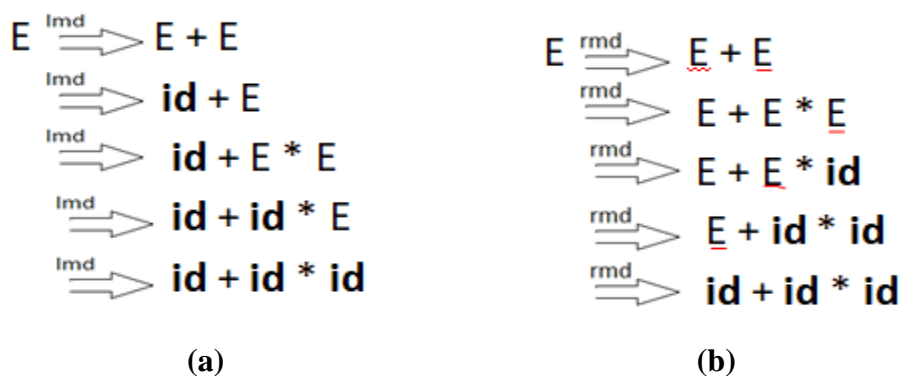


Fig 2.2 a) Leftmost derivation b) Rightmost derivation

Strings that appear in leftmost derivation are called left sentential forms. Strings that appear in rightmost derivation are called right sentential forms.

### Sentential Forms:

Given a grammar G with start symbol S, if  $S \Rightarrow \alpha$ , where  $\alpha$  may contain non-terminals or terminals, then  $\alpha$  is called the sentential form of G.

### Parse Tree:

A parse tree is a graphical representation of a derivation sequence of a sentential form.

In a parse tree:

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

Derive the string  $-(id + id)$

$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id$$

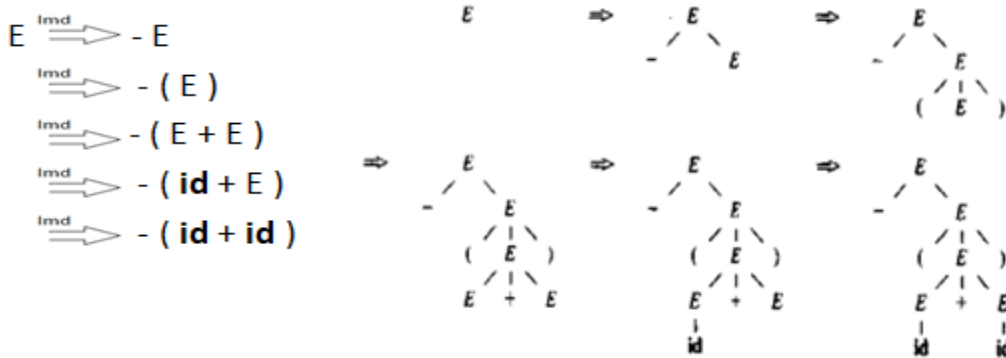


Fig 2.3 Build the parse tree for string  $-(id+id)$  from the derivation

**Yield or frontier of tree:**

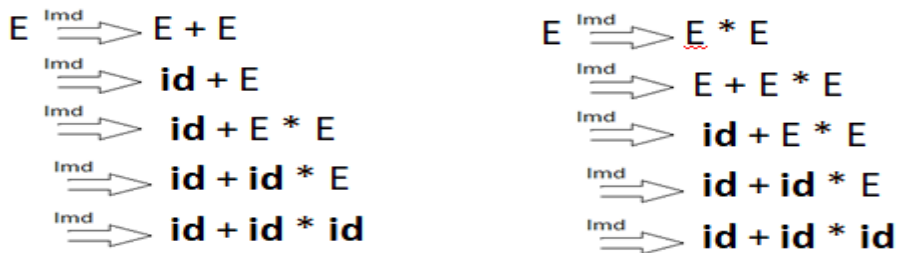
Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentential forms that are read from left to right. The sentential form in the parse tree is called yield or frontier of the tree.

**Ambiguity:**

A grammar that produces more than one parse tree for some sentence is said to be ambiguous grammar. i.e. An ambiguous grammar is one that produce more than one leftmost or more than one rightmost derivation for the same sentence.

Example : Given grammar  $G : E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id$

The sentence  $id+id*id$  has the following two distinct leftmost derivations:



The two corresponding parse trees are:



Fig 2.4 Two Parse tree for  $id+id*id$

Consider another example,

**stmt** → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

This grammar is ambiguous since the string **if** *E1* **then** **if** *E2* **then** *S1* **else** *S2* has the following

Two parse trees for leftmost derivation :

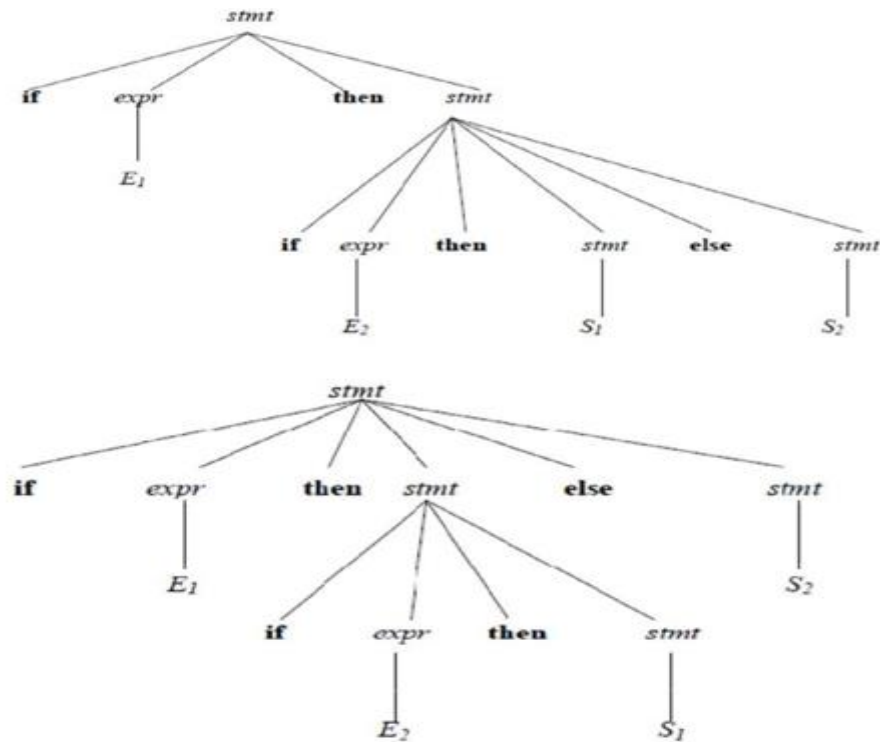


Fig 2.5 Two Parse tree for **if** *E1* **then** **if** *E2* **then** *S1* **else** *S2*

### Eliminating Ambiguity:

An ambiguous grammar can be rewritten to eliminate the ambiguity. e.g. Eliminate the ambiguity from “**dangling-else**” grammar,

**stmt** → **if** *expr* **then** *stmt*  
 | **if** *expr* **then** *stmt* **else** *stmt*  
 | **other**

Match each else with the closest previous unmatched then. This disambiguity rule can be incorporated into the grammar.

**stmt** → **matched\_stmt** | **unmatched\_stmt**  
**matched\_stmt** → **if** *expr* **then** **matched\_stmt** **else** **matched\_stmt**  
 | **other**  
**unmatched\_stmt** → **if** *expr* **then** *stmt*  
 | **if** *expr* **then** **matched\_stmt** **else** **unmatched\_stmt**

This grammar generates the same set of strings, but allows only one parsing for string.



**Table 2.1 Ambiguous grammar vs. Unambiguous grammar**

Ambiguous Grammar	Unambiguous Grammar
<p>A grammar is said to be ambiguous if for at least one string generated by it, it produces more than one-</p> <ul style="list-style-type: none"> <li>• parse tree</li> <li>• or derivation tree</li> <li>• or syntax tree</li> <li>• or leftmost derivation</li> <li>• or rightmost derivation</li> </ul>	<p>A grammar is said to be unambiguous if for all the strings generated by it, it produces exactly one-</p> <ul style="list-style-type: none"> <li>• parse tree</li> <li>• or derivation tree</li> <li>• or syntax tree</li> <li>• or leftmost derivation</li> <li>• or rightmost derivation</li> </ul>
<p>For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees.</p>	<p>For unambiguous grammar, leftmost derivation and rightmost derivation represents the same parse tree.</p>
<p>Ambiguous grammar contains less number of non-terminals.</p>	<p>Unambiguous grammar contains more number of non-terminals.</p>
<p>For ambiguous grammar, length of parse tree is less.</p>	<p>For unambiguous grammar, length of parse tree is large.</p>
<p>Ambiguous grammar is faster than unambiguous grammar in the derivation of a tree.  (Reason is above 2 points)</p>	<p>Unambiguous grammar is slower than ambiguous grammar in the derivation of a tree.</p>

**Removing Ambiguity by Precedence & Associativity Rules:**

An ambiguous grammar may be converted into an unambiguous grammar by implementing:

- Precedence Constraints
- Associativity Constraints

These constraints are implemented using the following rules:

**Rule-1:**

- The level at which the production is present defines the priority of the operator contained in it.
  - The higher the level of the production, the lower the priority of operator.
  - The lower the level of the production, the higher the priority of operator.

**Rule-2:**

- If the operator is left associative, induce left recursion in its production.
- If the operator is right associative, induce right recursion in its production.

**Example:** Consider the ambiguous Grammar:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid id$$

Introduce new variable / non-terminals at each level of precedence,

- an expression **E** for our example is a sum of one or more terms. (+, -)
- a term **T** is a product of one or more factors. (\*, /)
- a factor **F** is an identifier or parenthesised expression.

The resultant unambiguous grammar is:

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

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

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

Trying to derive the string **id+id\*id** using the above grammar will yield one unique derivation.

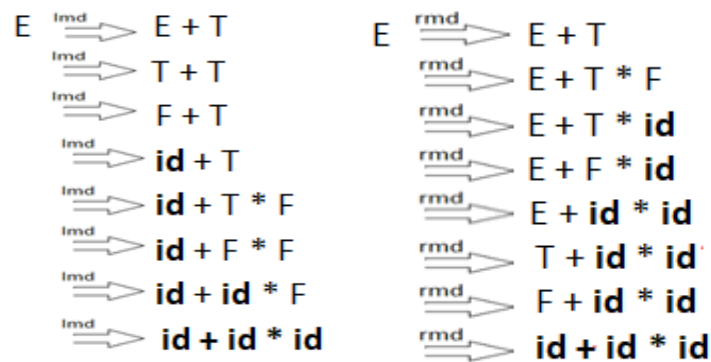


Fig 2.6 Distinct Leftmost and Rightmost derivation

### Regular Expression vs. Context Free Grammar:

- Every construct that can be described by a regular expression can be described by a grammar.
- NFA can be converted to a grammar that generates the same language as recognized by the NFA.
- Rules:
  - For each state  $i$  of the NFA, create a non-terminal symbol  $A_i$ .
  - If state  $i$  has a transition to state  $j$  on symbol  $a$ , introduce the production  $A_i \rightarrow a A_j$
  - If state  $i$  goes to state  $j$  on symbol  $\epsilon$ , introduce the production  $A_i \rightarrow A_j$
  - If  $i$  is an accepting state, introduce  $A_i \rightarrow \epsilon$
  - If  $i$  is the start state make  $A_i$  the start symbol of the grammar.

**Example:** The regular expression  $(a|b)^*abb$ , consider the NFA

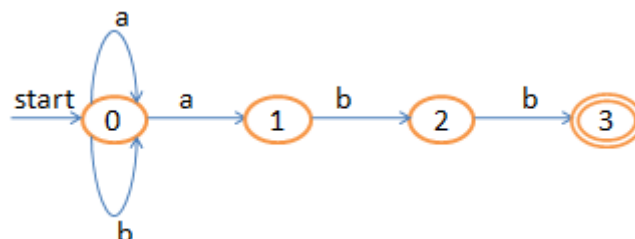


Fig 2.7 NFA for  $(a|b)^*abb$

Equivalent grammar is given by:

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

$$A_1 \rightarrow b A_2$$

$$A_2 \rightarrow b A_3$$

$$A_3 \rightarrow \varepsilon$$

### Types of Parser:

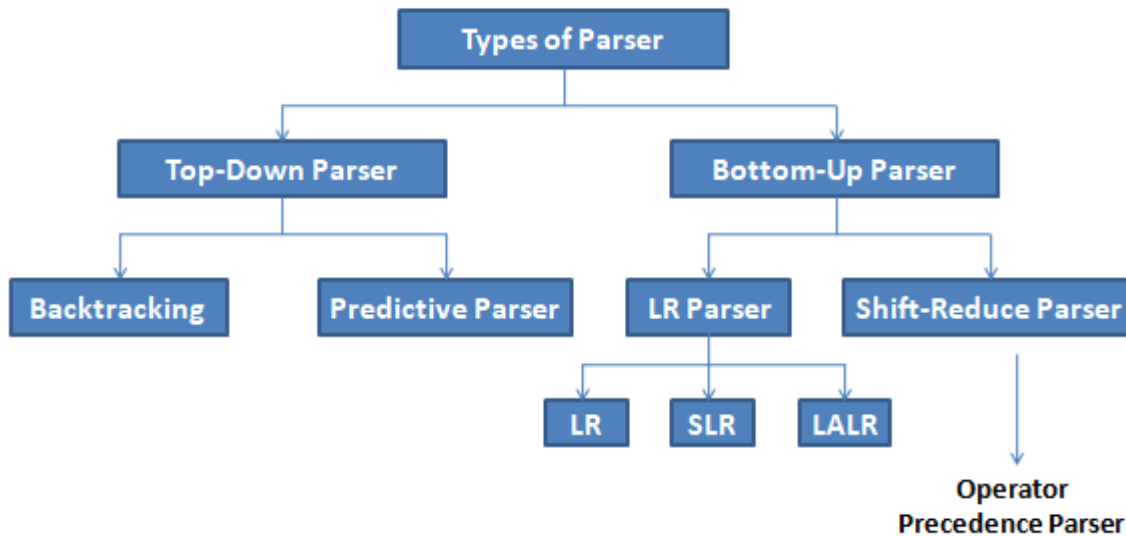


Fig 2.8 Types of Parser

### LR Parsing:

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.

### Why LR parsing:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

- The disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

### Bottom-Up Parsing:

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing. A general type of bottom-up parser is a shift-reduce parser.

### Shift-Reduce Parsing:

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

### Example:

Consider the grammar:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

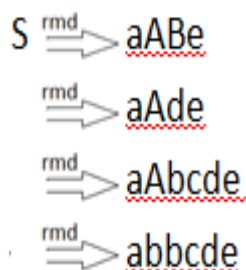
$B \rightarrow d$

The string to be recognized is abcde. We want to reduce the string to S.

### Steps of reduction:

abcde            (b,d can be reduced)  
 aAbcde        (leftmost b is reduced)  
 aAde           (now Abc,b,d qualified for reduction)  
 aABe           (d can be reduced)  
 S

Each replacement of the right side of a production by the left side in the above example is called reduction, which is equivalent to rightmost derivation in reverse.



### Handle:

A substring which is the right side of a production such that replacement of that substring by the production left side leads eventually to a reduction to the start symbol, by the reverse of a rightmost derivation is called a handle.

### Stack Implementation of Shift-Reduce Parsing:

There are two problems that must be solved if we are to parse by handle pruning. The first is to locate the substring to be reduced in a right-sentential form, and the second is to determine what production to choose in case there is more than one production with that substring on the right side.

A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string  $w$  to be parsed. 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
$\$$	$w\$$

The parser operates by shifting zero or more input symbols onto the stack until a handle is on top of the stack. 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
$\$ S$	$\$$

**Example:** The actions a shift-reduce parser in parsing the input string  $id_1+id_2*id_3$ , according to the ambiguous grammar for arithmetic expression.

	STACK	INPUT	ACTION
(1)	$\$$	$id_1 + id_2 * id_3 \$$	shift
(2)	$\$ id_1$	$+ id_2 * id_3 \$$	reduce by $E \rightarrow id$
(3)	$\$ E$	$+ id_2 * id_3 \$$	shift
(4)	$\$ E +$	$id_2 * id_3 \$$	shift
(5)	$\$ E + id_2$	$* id_3 \$$	reduce by $E \rightarrow id$
(6)	$\$ E + E$	$* id_3 \$$	shift
(7)	$\$ E + E *$	$id_3 \$$	shift
(8)	$\$ E + E * id_3$	$\$$	reduce by $E \rightarrow id$
(9)	$\$ E + E * E$	$\$$	reduce by $E \rightarrow E * E$
(10)	$\$ E + E$	$\$$	reduce by $E \rightarrow E + E$
(11)	$\$ E$	$\$$	accept

Fig 2.9 Configuration of Shift Reduce Parser on input  $id_1+id_2*id_3$

RIGHT-SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 + id_2 * id_3$	$id_1$	$E \rightarrow id$
$E + id_2 * id_3$	$id_2$	$E \rightarrow id$
$E + E * id_3$	$id_3$	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
$E$		

Fig 2.10 Reductions made by Shift Reduce Parser

While the primary operations of the parser are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

(1) shift, (2) reduce, (3) accept, and (4) error.

- In a **shift** action, the next input symbol is shifted unto the top of the stack.
- In a **reduce** action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.
- In an **accept** action, the parser announces successful completion of parsing.
- In an **error** action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

Figure 2.11 represents the stack implementation of shift reduce parser using unambiguous grammar.

Stack	Input	Action
S	id+id*id\$	shift
Sid	+id*id\$	reduce by $F \rightarrow id$
SF	+id*id\$	reduce by $T \rightarrow F$
ST	+id*id\$	reduce by $E \rightarrow T$
SE	+id*id\$	shift
SE+	id*id\$	shift
SE+id	*id\$	reduce by $F \rightarrow id$
SE+F	*id\$	reduce by $T \rightarrow F$
SE+T	*id\$	shift
SE+T*	id\$	shift
SE+T*id	\$	reduce by $F \rightarrow id$
SE+T*F	\$	reduce by $T \rightarrow T*F$
SE+T	\$	reduce by $E \rightarrow E+T$
SE	\$	accept

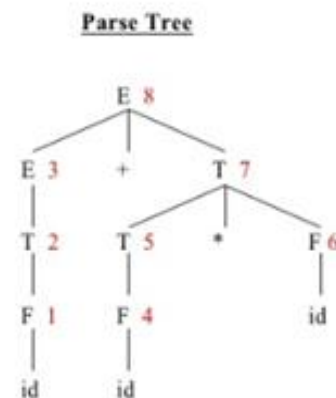


Fig 2.11 A stack implementation of a Shift-Reduce parser

### Operator Precedence Parsing:

Operator grammars have the property that no production right side is  $\epsilon$  (empty) or has two

adjacent non terminals. This property enables the implementation of efficient operator-precedence parsers.

**Example:** The following grammar for expressions:

$$E \rightarrow E A E \mid (E) \mid -E \mid \text{id}$$

$$A \rightarrow + \mid - \mid * \mid / \mid ^$$

This is not an operator grammar, because the right side EAE has two consecutive non-terminals. However, if we substitute for A each of its alternate, we obtain the following operator grammar:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid ( E ) \mid E ^ E \mid - E \mid \text{id}$$

In operator-precedence parsing, we define three disjoint precedence relations between pair of terminals. This parser relies on the following three precedence relations.

Relation	Meaning
$a < \cdot b$	$a$ yields precedence to $b$
$a \cdot \hat{=} b$	$a$ has the same precedence as $b$
$a \cdot > b$	$a$ takes precedence over $b$

Fig 2.12 Precedence Relations

These precedence relations guide the selection of handles. These operator precedence relations allow delimiting the handles in the right sentential forms:  $< \cdot$  marks the left end,  $\cdot \hat{=}$  appears in the interior of the handle, and  $\cdot >$  marks the right end.

	id	+	*	\$
id		$\cdot >$	$\cdot >$	$\cdot >$
+	$< \cdot$	$\cdot >$	$< \cdot$	$\cdot >$
*	$< \cdot$	$\cdot >$	$\cdot >$	$\cdot >$
\$	$< \cdot$	$< \cdot$	$< \cdot$	$\cdot >$

Fig 2.13 Operator Precedence Relation Table

**Example:** The input string:  $\text{id}_1 + \text{id}_2 * \text{id}_3$

After inserting precedence relations the string becomes:

$$\$ < \cdot \text{id}_1 \cdot > + < \cdot \text{id}_2 \cdot > * < \cdot \text{id}_3 \cdot > \$$$

Having precedence relations allows identifying handles as follows:

1. Scan the string from left end until the leftmost  $\cdot >$  is encountered.
2. Then scan backwards over any  $\hat{=}$ 's until a  $< \cdot$  is encountered.
3. Everything between the two relations  $< \cdot$  and  $\cdot >$  forms the handle.

Stack	Rule	Input	Comments
\$ < id > + < id > * < id > \$	$E \rightarrow id$	\$ id + id * id \$	Here the first "id" is looked as the handle and since we were able to reduce, we reduce it in the input
\$ < + < id > * < id > \$	$E \rightarrow id$	\$ E + id * id \$	The second handle is also "id" since that is available between a pair of lesser than and greater than precedences
\$ < + < * < id > \$	$E \rightarrow id$	\$ E + E * id \$	The third handle is also "id".
\$ < + < * > \$	$E \rightarrow E * E$	\$ E + E * E \$	The fourth handle is $E * E$ , and is popped in the stack and we push the greater than symbol.
\$ < + > \$	$E \rightarrow E + E$	\$ E + E \$	The last handle is $E + E$ and that is also reduced.
\$ \$			The stack is empty and has only the \$ symbol, we say the string is accepted.

### Defining Precedence Relations:

The precedence relations are defined using the following rules:

#### Rule-01:

- If precedence of b is higher than precedence of a, then we define  $a < b$
- If precedence of b is same as precedence of a, then we define  $a = b$
- If precedence of b is lower than precedence of a, then we define  $a > b$

#### Rule-02:

- An identifier is always given the higher precedence than any other symbol.
- \$ symbol is always given the lowest precedence.

#### Rule-03:

- If two operators have the same precedence, then we go by checking their associativity.
  1.  $\dagger$  is of highest precedence and right-associative,
  2.  $*$  and  $/$  are of next highest precedence and left-associative, and
  3.  $+$  and  $-$  are of lowest precedence and left-associative,

	+	-	*	/	$\dagger$	id	(	)	\$
+	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
-	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
*	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
/	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
$\dagger$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
id	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$			$\cdot >$	$\cdot >$
(	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot =$	
)	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$			$\cdot >$	
\$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$		$\cdot >$

Fig 2.14 Operator Precedence Relation Table



STACK	INPUT	COMMENT
\$	< id+id*id \$	shift id
\$ id	> +id*id \$	pop the top of the stack id
\$	< +id*id \$	shift +
\$ +	< id*id \$	shift id
\$ +id	> *id \$	pop id
\$ +	< *id \$	shift *
\$ +*	< id \$	shift id
\$ +* id	> \$	pop id
\$ +*	> \$	pop *
\$ +	> \$	pop +
\$	\$	accept

Fig 2.15 Stack Implementation

### Implementation of Operator-Precedence Parser:

- An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR(1) grammars.
- More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive non-terminals and epsilon never appear in the right-hand side of any rule.

### Steps involved in Parsing:

1. Ensure the grammar satisfies the pre-requisite.
2. Computation of the function LEADING()
3. Computation of the function TRAILING()
4. Using the computed leading and trailing ,construct the operator Precedence Table
5. Parse the given input string based on the algorithm
6. Compute Precedence Function and graph.

### Computation of LEADING:

- Leading is defined for every non-terminal.
- Terminals that can be the first terminal in a string derived from that non-terminal.
- $LEADING(A) = \{ a \mid A \Rightarrow^+ \gamma a \delta \}$ , where  $\gamma$  is  $\epsilon$  or any non-terminal,  $\Rightarrow^+$  indicates derivation in one or more steps, A is a non-terminal.

**Algorithm for LEADING(A):**

```

{
1. 'a' is in LEADING(A) is  $A \rightarrow \gamma a \delta$  where  $\gamma$  is  $\epsilon$  or any non-terminal.
2.If 'a' is in LEADING(B) and  $A \rightarrow B$ , then 'a' is in LEADING(A).
}

```

**Computation of TRAILING:**

- Trailing is defined for every non-terminal.
- Terminals that can be the last terminal in a string derived from that non-terminal.
- $TRAILING(A) = \{ a \mid A \Rightarrow^+ \gamma a \delta \}$ , where  $\delta$  is  $\epsilon$  or any non-terminal,  $\Rightarrow^+$  indicates derivation in one or more steps, A is a non-terminal.

**Algorithm for TRAILING(A):**

```

{
1. 'a' is in TRAILING(A) is  $A \rightarrow \gamma a \delta$  where  $\delta$  is  $\epsilon$  or any non-terminal.
2.If 'a' is in TRAILING(B) and  $A \rightarrow B$ , then 'a' is in TRAILING(A).
}

```

**Example 1:** Consider the unambiguous grammar,

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow \text{id}$$

**Step 1:** Compute LEADING and TRAILING:

$$LEADING(E) = \{ +, LEADING(T) \} = \{ +, *, (, \text{id} \}$$

$$LEADING(T) = \{ *, LEADING(F) \} = \{ *, (, \text{id} \}$$

$$LEADING(F) = \{ (, \text{id} \}$$

$$TRAILING(E) = \{ +, TRAILING(T) \} = \{ +, *, ), \text{id} \}$$

$$TRAILING(T) = \{ *, TRAILING(F) \} = \{ *, ), \text{id} \}$$

$$TRAILING(F) = \{ ), \text{id} \}$$

**Step 2:** After computing LEADING and TRAILING, the table is constructed between all the terminals in the grammar including the '\$' symbol.

```

for each production  $A \rightarrow X_1 X_2 X_3 \dots X_n$ 
  for  $i=1$  to  $n-1$ 
    1. if  $X_i$  and  $X_{i+1}$  are terminals
       set  $X_i \doteq X_{i+1}$ 
    2. if  $i \leq n-2$  and  $X_i$  and  $X_{i+2}$  are terminals and  $X_{i+1}$  is a non-terminal,
       set  $X_i \doteq X_{i+2}$ 
    3. if  $X_i$  is a terminal and  $X_{i+1}$  is a non-terminal ,then for all 'a' in
       LEADING( $X_{i+1}$ )
       set  $X_i \prec a$ 
    4. if  $X_i$  is a non-terminal and  $X_{i+1}$  is a terminal ,then for all 'a' in
       TRAILING( $X_i$ )
       set  $a \succ X_{i+1}$ 
    5. Set  $\$ \prec \text{Leading}(S)$  and  $\text{Trailing}(S) \succ \$$ , where  $S$ -start symbol.

```

Fig 2.16 Algorithm for constructing Precedence Relation Table

	+	*	id	(	)	\$
+	>	<	<	<	>	>
*	>	>	<	<	>	>
id	>	>	e	e	>	>
(	<	<	<	<	=	e
)	>	>	e	e	>	>
\$	<	<	<	<	e	Accept

Fig 2.17 Precedence Relation Table

**Step 3:** Parse the given input string (id+id)\*id\$

```

Set ip to point to the first symbol of w$
Repeat forever
  if $ is on the top of the stack and ip points to $ then return
  else begin
    Let a be the top terminal on the stack, and b the symbol pointed to by ip
    if  $a < b$  or  $a = b$  then
      push b onto the stack
      advance ip to the next input symbol
    end
    else if  $a > b$  then
      repeat
        pop the stack
      until the top stack terminal is related by  $<$ 
      to the terminal most recently popped
    else error()
  end
end

```

Fig 2.18 Parsing Algorithm

STACK	REL.	INPUT	ACTION
\$	\$ < (	(id+id)*id\$	Shift (
\$(	( < id	id+id)*id\$	Shift id
\$( id	id > +	+id)*id\$	Pop id
\$(	( < +	+id)*id\$	Shift +
\$(+	+ < id	id)*id\$	Shift id
\$(+id	id > )	)*id\$	Pop id
\$(+	+ > )	)*id\$	Pop +
\$(	( = )	)*id\$	Shift )
\$( )	) > *	*id \$	Pop )
\$(			Pop (
\$	\$ < *	*id \$	Shift *
\$*	* < id	id\$	Shift id
\$*id	id > \$	\$	Pop id
\$*	* > \$	\$	Pop *
\$		\$	Accept

Fig 2.19 Parse the input string (id+id)\*id\$

### Precedence Functions:

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two precedence functions  $f$  and  $g$  that map terminal symbols to integers. We attempt to select  $f$  and  $g$  so that, for symbols  $a$  and  $b$ .

1.  $f(a) < g(b)$  whenever  $a \cdot b$ .
2.  $f(a) = g(b)$  whenever  $a = b$ . and
3.  $f(a) > g(b)$  whenever  $a \cdot > b$ .

### Algorithm for Constructing Precedence Functions:

1. Create functions  $f_a$  for each grammar terminal  $a$  and for the end of string symbol.
2. Partition the symbols in groups so that  $f_a$  and  $g_b$  are in the same group if  $a = b$  (there can be symbols in the same group even if they are not connected by this relation).

3. Create a directed graph whose nodes are in the groups, next for each symbols  $a$  and  $b$  do: place an edge from the group of  $g_b$  to the group of  $f_a$  if  $a < b$ , otherwise if  $a > b$  place an edge from the group of  $f_a$  to that of  $g_b$ .
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of  $f_a$  and  $g_b$  respectively.

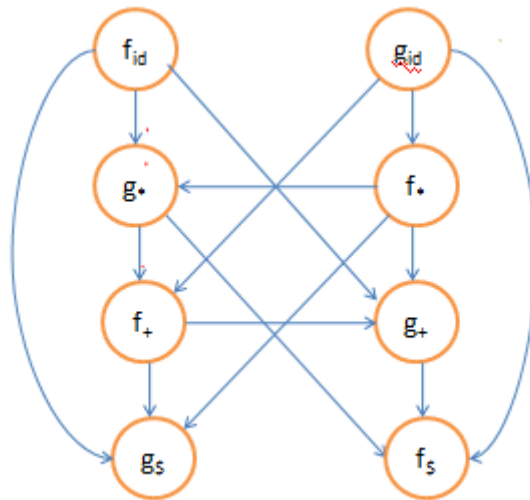


Fig 2.20 Precedence Graph

There are no cycles, so precedence function exist. As  $f\$$  and  $g\$$  have no out edges,  $f(\$) = g(\$) = 0$ . The longest path from  $g_+$  has length 1, so  $g(+)=1$ . There is a path from  $g_{id}$  to  $f^*$  to  $g^*$  to  $f_+$  to  $g_+$  to  $f\$$ , so  $g(id)=5$ . The resulting precedence functions are:

	id	+	*	\$
f	4	2	4	0
g	5	1	3	0

**Example 2:**

Consider the following grammar, and construct the operator precedence parsing table and check whether the input string (i)  $*id=id$  (ii)  $id*id=id$  are successfully parsed or not?

$$S \rightarrow L=R$$

$$S \rightarrow R$$

$$L \rightarrow *R$$

$$L \rightarrow id$$

$$R \rightarrow L$$

**Solution:**

**1. Computation of LEADING:**

LEADING(S) = {=, \*, id}

LEADING(L) = {\*, id}

LEADING(R) = {\*, id}

**2.Computation of TRAILING:**

TRAILING(S) = {=, \*, id}

TRAILING(L)= {\*, id}

TRAILING(R)= {\*, id}

**3.Precedence Table:**

	=	*	id	\$
=	e	<.	<.	.>
*	.>	<.	<.	.>
id	.>	e	e	.>
\$	<.	<.	<.	accept

\* All undefined entries are error (e).

**4.Parsing the given input string:**

**1. \*id = id**

STACK	INPUT STRING	ACTION
\$	*id=id\$	\$<.* Push
\$*	id=id\$	*<.id Push
\$*id	=id\$	id.>= Pop
\$*	=id\$	*.>= Pop
\$	=id\$	\$<.= Push
\$=	id\$	=<.id Push
\$=id	\$	id.>\$ Pop
\$=	\$	=.>\$ Pop
\$	\$	Accept

**2. id\*id=id**

STACK	INPUT STRING	ACTION
\$	id*id=id\$	\$<.idPush
\$id	*id=id\$	Error

**Example 3:** Check whether the following Grammar is an operator precedence grammar or not.

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

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

**Solution:**

**1. Computation of LEADING:**

$$\text{LEADING}(E) = \{+, *, \text{id}\}$$

**2. Computation of TRAILING:**

$$\text{TRAILING}(E) = \{+, *, \text{id}\}$$

**3. Precedence Table:**

	+	*	id	\$
+	<./.>	<./.>	<.	.>
*	<./.>	<./.>	<.	.>
id	.>	.>		.>
\$	<.	<.	<.	

All undefined entries are error. Since the precedence table has multiple defined entries, the grammar is not an operator precedence grammar.

**Top-Down Parsing- Recursive Descent Parsing:**

Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string. Equivalently it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

A general form top-down parsing called recursive descent parsing, involves backtracking, that is making repeated scans of the input. A special case of recursive descent parsing called predictive parsing, where no backtracking is required.

Consider the grammar

$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

and the input string  $w=cad$ . Construction of parse is shown in fig 2.21.

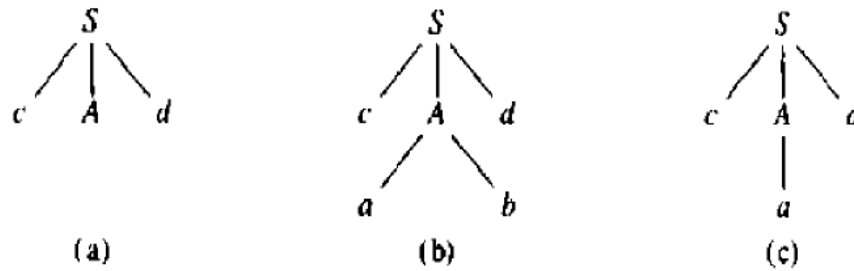


Fig 2.21 Steps in Top-down Parse

The leftmost leaf, labeled  $c$ , matches the first symbol of  $w$ , hence advance the input pointer to  $a$ , the second symbol of  $w$ . Fig 2.21(b) and (c) shows the backtracking required to match the input string.

**Predictive Parser:**

A grammar after eliminating left recursion and left factoring can be parsed by a recursive descent parser that needs no backtracking is called a predictive parser. Let us understand how to eliminate left recursion and left factoring.

**Eliminating Left Recursion:**

A grammar is said to be left recursive if it has a non-terminal  $A$  such that there is a derivation  $A \Rightarrow A\alpha$  for some string  $\alpha$ . Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production  $A \rightarrow A\alpha \mid \beta$  it can be replaced with a sequence of two productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Without changing the set of strings derivable from  $A$ .

**Example :** Consider the following grammar for arithmetic expressions:

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

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

$$F \rightarrow (E) \mid id$$

First eliminate the left recursion for  $E$  as

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

Then eliminate for  $T$  as

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

Thus the obtained grammar after eliminating left recursion is

$$E \rightarrow TE'$$



$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid \text{id}$$
**Algorithm to eliminate left recursion:**

1. Arrange the non-terminals in some order  $A_1, A_2 \dots A_n$ .
2. **for**  $i := 1$  to  $n$  do begin
  - for**  $j := 1$  to  $i-1$  do begin
    - replace each production of the form  $A_i \rightarrow A_j \gamma$
    - by the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ .
    - where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the current  $A_j$ -productions;
  - end**
  - eliminate the immediate left recursion among the  $A_i$ - productions
  - end**

**Left factoring:**

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal  $A$ , we can rewrite the  $A$ -productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ , it can be rewritten as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

Consider the grammar,

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

Here,  $i, t, e$  stand for **if**, **the**, and **else** and  $E$  and  $S$  for “expression” and “statement”.

After Left factored, the grammar becomes

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$E \rightarrow b$$
**Non-recursive Predictive Parsing:**

It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of

determining the production to be applied for a non-terminal. The non-recursive parser in Fig 2.22 looks up the production to be applied in a parsing table.

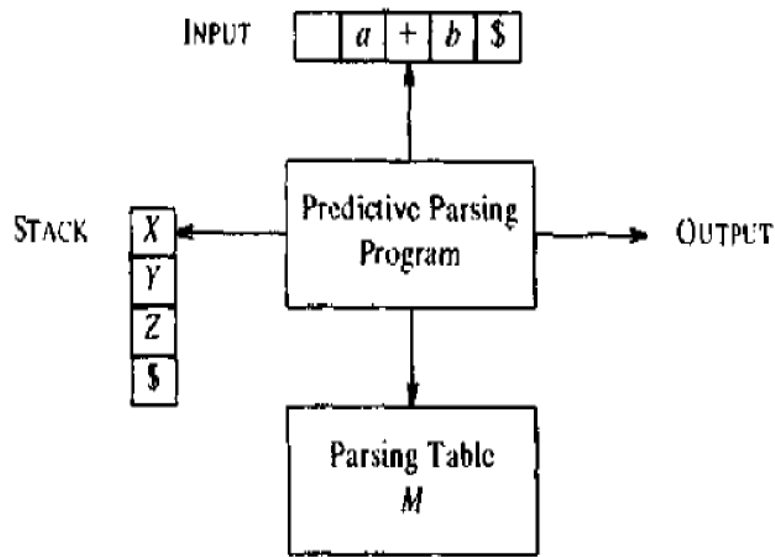


Fig 2.22 Model of a Non-recursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by \$, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with \$ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of S. The parsing table is a two-dimensional array  $M[A,a]$ , where  $A$  is a non-terminal, and  $a$  is a terminal or the symbol \$.

The program considers  $X$ , the symbol on top of the stack, and  $a$ , the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1. If  $X = a = \$$ , the parser halts and announces successful completion of parsing.
2. If  $X = a \neq \$$ , the parser pops  $X$  off the stack and advances the input pointer to the next input symbol.
3. If  $X$  is a nonterminal, the program consults entry  $M[X,a]$  of the parsing table  $M$ . This entry will be either an  $X$ -production of the grammar or an error entry. If, for example,  $M[X,a] = \{X \rightarrow UVW\}$ , the parser replaces  $X$  on top of the stack by  $WVU$  (with  $U$  on top). If  $M[X, a] = \text{error}$ , the parser calls an error recovery routine.

### Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar

G. These functions are FIRST and FOLLOW.

**Rules for FIRST():**

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

**Rules for FOLLOW():**

1. If S is a start symbol, then FOLLOW(S) contains \$.
2. If there is a production  $A \rightarrow \alpha B \beta$ , then everything in FIRST( $\beta$ ) except  $\epsilon$  is placed 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).

**Algorithm for construction of predictive parsing table:**

Input : Grammar G

Output : Parsing table M

Method :

1. For each production  $A \rightarrow \alpha$  of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST( $\alpha$ ), add  $A \rightarrow \alpha$  to  $M[A, a]$ .
3. If  $\epsilon$  is in FIRST( $\alpha$ ), add  $A \rightarrow \alpha$  to  $M[A, b]$  for each terminal b in FOLLOW(A). If  $\epsilon$  is in FIRST( $\alpha$ ) and \$ is in FOLLOW(A), add  $A \rightarrow \alpha$  to  $M[A, \$]$ .
4. Make each undefined entry of M be error.

**Algorithm : Non-recursive predictive parsing.**

**Input:** A string w and a parsing table M for grammar G.

**Output:** If w is in L(G), a leftmost derivation of w; otherwise, an error .

**Method:** Initially, the parser is in a configuration in which it has \$\$ on the stack with S, the start symbol of G on top, and w\$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input.

set ip to point to the first symbol of w\$:

**repeat**

let X be the top stack symbol and a the symbol pointed to by ip;

**if** X is a terminal or \$ **then**

**if** X = a **then**

pop X from the stack and advance ip

**else** error()

**else** /\* X is a nonterminal \*/

**if**  $M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$ , **then**

**begin**

pop X from the stack:

push  $Y_k, Y_{k-1} \dots Y_1$ , onto the stack, with  $Y_1$  on top;

output the production  $X \rightarrow Y_1 Y_2 \dots Y_k$

**end**

**else** error()

**until**  $X \neq \$$  /\* stack is empty\*/

**Example:**

Consider the following grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

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

After eliminating left recursion the grammar is

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

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

**FIRST() :**

$\text{FIRST}(E) = \{ (, \text{id} \}$

$\text{FIRST}(E') = \{ +, \epsilon \}$

$\text{FIRST}(T) = \{ (, \text{id} \}$

$\text{FIRST}(T') = \{ *, \epsilon \}$

$\text{FIRST}(F) = \{ (, \text{id} \}$

**FOLLOW():**

FOLLOW(E) = { \$, ) }

FOLLOW(E') = { \$, ) }

FOLLOW(T) = { +, \$, ) }

FOLLOW(T') = { +, \$, ) }

FOLLOW(F) = { +, \*, \$, , ) }

Predictive parsing table for the given grammar is shown in Fig 2.23.

M[X,a]	id	+	*	(	)	\$
E	E →TE'			E →TE'		
E'		E' →+TE'			E' →ε	E' →ε
T	T →FT'			T →FT'		
T'		T' →ε	T' →*FT'		T' →ε	T' →ε
F	F →id			F → (E)		

Fig 2.23 Parsing table

With input **id+id\*id** the predictive parser makes the sequence of moves shown in Fig 2,24.

STACK	INPUT	OUTPUT
\$E	<b>id+id*id\$</b>	E →TE'
\$E'T	<b>id+id*id\$</b>	T →FT'
\$E'T'F	<b>id+id*id\$</b>	F →id
\$E'T'id	<b>id+id*id\$</b>	pop
\$E'T'	<b>+id*id\$</b>	T' →ε
\$E'	<b>+id*id\$</b>	E' →+TE'
\$E'T+	<b>+id*id\$</b>	pop
\$E'T	<b>id*id\$</b>	T →FT'
\$E'T'F	<b>id*id\$</b>	F →id
\$E'T'id	<b>id*id\$</b>	Pop
\$E'T'	<b>*id\$</b>	T' →*FT'
\$E'T'F*	<b>*id\$</b>	Pop
\$E'T'F	<b>id\$</b>	F →id
\$E'T'id	<b>id\$</b>	Pop
\$E'T'	<b>\$</b>	T' →ε
\$E'	<b>\$</b>	E' →ε
\$	<b>\$</b>	Accept

Fig 2.24 Moves made by predictive parser on input **id+id\*id**

### LL(1) Grammars:

For some grammars the parsing table may have some entries that are multiply-defined. For example, if  $G$  is left recursive or ambiguous, then the table will have at least one multiply-defined entry. A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

**Example:** Consider this following grammar:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

After eliminating left factoring, we have

$$S \rightarrow iEtSS' \mid a \quad S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

$$\text{FIRST}(S) = \{ i, a \}$$

$$\text{FIRST}(S') = \{ e, \epsilon \}$$

$$\text{FIRST}(E) = \{ b \}$$

$$\text{FOLLOW}(S) = \{ \$, e \}$$

$$\text{FOLLOW}(S') = \{ \$, e \}$$

$$\text{FOLLOW}(E) = \{ t \}$$

Parsing Table for the grammar:

NON-TERMINAL	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Since there are more than one production for an entry in the table, the grammar is not LL(1) grammar.

### LR PARSERS:

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is 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 lookahead that are used in making parsing decisions.. When (k) is omitted, it is assumed to be 1. Table 2.2 shows the comparison between LL and LR parsers.

**Table 2.2 LL vs. LR**

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

**Types of LR parsing method:**

1. SLR- Simple LR
  - Easiest to implement, least powerful.
2. CLR- Canonical LR
  - Most powerful, most expensive.
3. LALR- Look -Ahead LR
  - Intermediate in size and cost between the other two methods

**The LR Parsing Algorithm:**

The schematic form of an LR parser is shown in Fig 2.25. 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 parser. The parsing table alone 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  $s_0X_1s_1X_2s_2\dots\dots X_ms_m$ , where  $s_m$  is on top. Each  $X_i$  is a grammar symbol and each  $s_i$  is a symbol called a state.

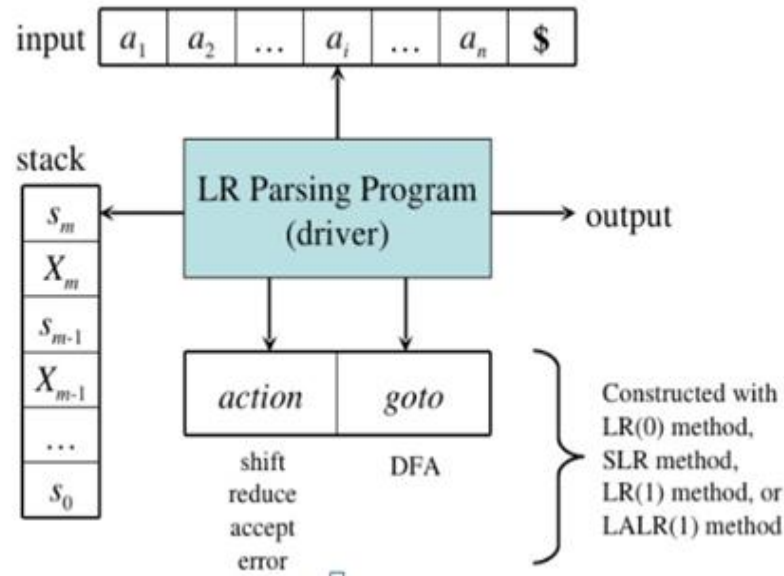


Fig 2.25 Model of an LR Parser

The parsing table consists of two parts : *action* and *goto* functions.

**Action** : The parsing program determines  $s_m$ , the state currently on top of stack, and  $a_i$ , the current input symbol. It then consults  $action[s_m, a_i]$  in the action table 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.

**Goto** : The function *goto* takes a state and grammar symbol as arguments and produces a state.

### CONSTRUCTING SLR PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute  $goto(I, X)$ , where,  $I$  is set of items and  $X$  is grammar symbol.

### LR(0) items:

An *LR(0) item* of a grammar  $G$  is a production of  $G$  with a dot at some position of the right side. For example, production  $A \rightarrow XYZ$  yields the four items :

- $A \rightarrow \bullet XYZ$
- $A \rightarrow X \bullet YZ$
- $A \rightarrow XY \bullet Z$



$$A \rightarrow XYZ \bullet$$

**Closure operation:**

If  $I$  is a set of items for a grammar  $G$ , then  $\text{closure}(I)$  is the set of items constructed from  $I$  by the two rules:

1. Initially, every item in  $I$  is added to  $\text{closure}(I)$ .
2. If  $A \rightarrow \alpha \cdot B\beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production, then add the item  $B \rightarrow \cdot \gamma$  to  $I$ , if it is not already there. We apply this rule until no more new items can be added to  $\text{closure}(I)$ .

**Goto operation:**

$\text{Goto}(I, X)$  is defined to be the closure of the set of all items  $[A \rightarrow \alpha X \bullet \beta]$  such that  $[A \rightarrow \alpha \bullet X \beta]$  is in  $I$ . Steps to construct SLR parsing table for grammar  $G$  are:

1. Augment  $G$  and produce  $G'$
2. Construct the canonical collection of set of items  $C$  for  $G'$
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires  $\text{FOLLOW}(A)$  for each non-terminal of grammar.

**Algorithm for construction of 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 functions for state  $i$  are determined as follows:
  - (a) If  $[A \rightarrow \alpha \bullet a \beta]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to “shift  $j$ ”. Here  $a$  must be terminal.
  - (b) If  $[A \rightarrow \alpha \bullet]$  is in  $I_i$ , then set  $\text{action}[i, a]$  to “reduce  $A \rightarrow \alpha$ ” for all  $a$  in  $\text{FOLLOW}(A)$ .
  - (c) If  $[S' \rightarrow \bullet S]$  is in  $I_i$ , then set  $\text{action}[i, \$]$  to “accept”.

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state  $i$  are constructed for all non-terminals  $A$  using the rule: If  $\text{goto}(I_i, A) = I_j$ , then  $\text{goto}[i, A] = j$ .
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  $[S' \rightarrow \bullet S]$ .

**SLR Parsing algorithm:**

**Input:** An input string  $w$  and an LR parsing table with functions *action* and *goto* for grammar  $G$ .

**Output:** If  $w$  is in  $L(G)$ , a bottom-up-parse for  $w$ ; otherwise, an error indication.

**Method:** Initially, the parser has  $s_0$  on its stack, where  $s_0$  is the initial state, and  $w\$$  in the input buffer. The parser then executes the following program :

set  $ip$  to point to the first input symbol of  $w\$$ ;

**repeat forever begin**

let  $s$  be the state on top of the stack and  $a$  the symbol pointed to by  $ip$ ;

**if**  $action[s, a] = \text{shift } s''$  **then begin**

push  $a$  then  $s''$  on top of the stack;

advance  $ip$  to the next input symbol

**end**

**else if**  $action[s, a] = \text{reduce } A \rightarrow \beta$  **then begin**

pop  $2 * |\beta|$  symbols off the stack;

let  $s''$  be the state now on top of the stack;

push  $A$  then  $goto[s'', A]$  on top of the stack;

output the production  $A \rightarrow \beta$

**end**

**else if**  $action[s, a] = \text{accept}$  **then**

**return**

**else**  $error()$

**end**

Example: Implement SLR Parser for the given grammar:

1.  $E \rightarrow E + T$

2.  $E \rightarrow T$

3.  $T \rightarrow T * F$

4.  $T \rightarrow F$

5.  $F \rightarrow (E)$

6.  $F \rightarrow id$

**Step 1 :** Convert given grammar into augmented grammar.

**Augmented grammar:**

$E' \rightarrow E$

$E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow T * F$   
 $T \rightarrow F$   
 $F \rightarrow (E)$   
 $F \rightarrow id$

**Step 2 : Find LR (0) items.**

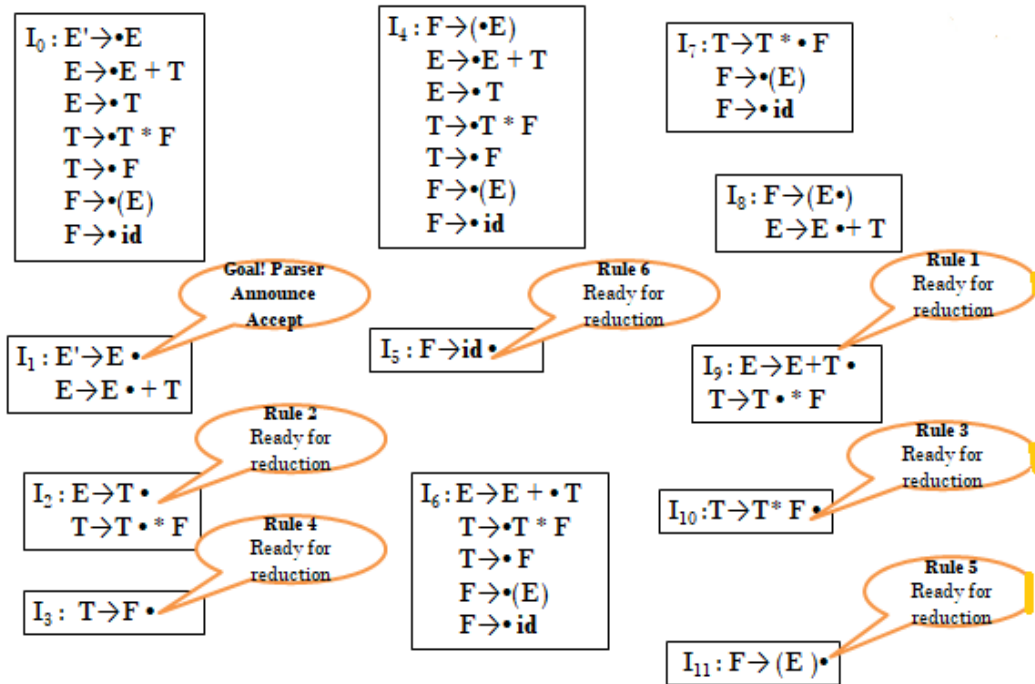


Fig 2.26 Canonical LR(0) collections

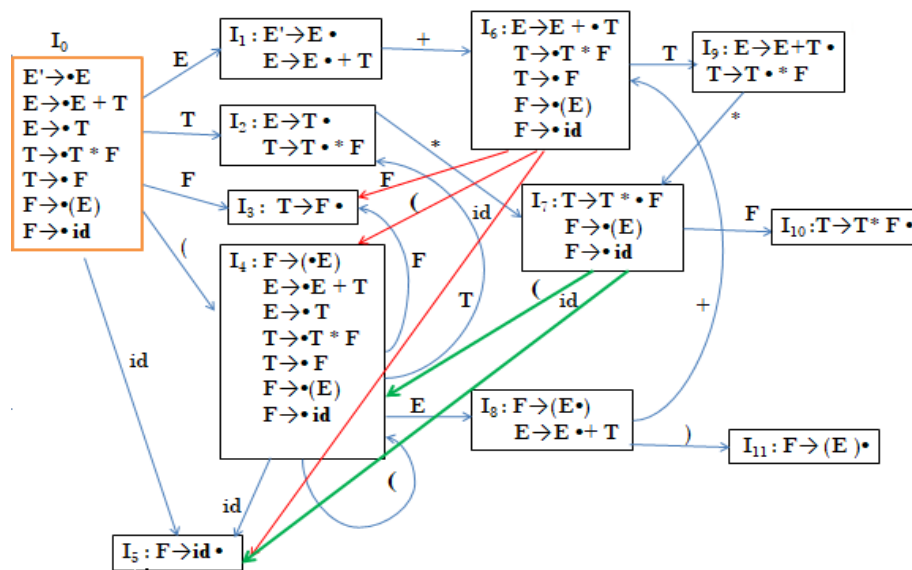


Fig 2.27 DFA representing the GOTO on symbols

**Step 3 :** Construction of Parsing table.

1. Computation of FOLLOW is required to fill the reduction action in the ACTION part of the table.

FOLLOW(E) = {+,),\$ }

FOLLOW(T) = {\*,+),,\$}

FOLLOW(F) = {\*,+),,\$}

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			

Fig 2.28 Parsing Table for the expression grammar

1.  $s_i$  means shift and stack state  $i$ .
2.  $r_j$  means reduce by production numbered  $j$ .
3. acc means accept.
4. Blank means error.

**Step 4:** Parse the given input. The Fig 2.29 shows the parsing the string  $id*id+id$  using stack implementation.

Stack	Input	Action
0	id*id+id\$	s5 Shift 5
0 id 5	*id+id\$	r6 Reduce by $F \rightarrow id$
0 F 3	*id+id\$	r4 Reduce by $T \rightarrow F$
0 T 2	*id+id\$	s7 Shift 7
0 T 2 * 7	id+id\$	s5 Shift 5
0 T 2 * 7 id 5	+id\$	r6 Reduce by $F \rightarrow id$
0 T 2 * 7 F 10	+id\$	r3 Reduce by $T \rightarrow T * F$
0 T 2	+id\$	r2 Reduce by $E \rightarrow T$
0 E 1	+id\$	s6 Shift 6
0 E 1 + 6	id\$	s5 Shift 5
0 E 1 + 6 id 5	\$	r6 Reduce by $F \rightarrow id$
0 E 1 + 6 F 3	\$	r4 Reduce by $T \rightarrow F$
0 E 1 + 6 T 9	\$	r1 Reduce by $E \rightarrow E + T$
0 E 1	\$	Accept

Fig 2.29 Moves of LR parser on **id\*id+id**