UNIT II

SYNTAX ANALYSIS

2.1 INTRODUCTION

Every programming language has rules that prescribe the syntactic structure of well formed programs.

The syntax of programming language constructs can be described by context free CFG grammars or BNF-(Backus Naur Form).

Notation

- A Grammar gives a precise yet easy to understand syntactic specification of a programming language.
- From grammars we can automatically construct an efficient parser, that determines if a source programs is syntactically well formed.
- A properly designed grammar, is useful for the translation of source programs into correct object code and for the detection of errors.
- 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 using the grammatical description of the language.

2.2 THE ROLE OF THE PARSER

- The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar, for the source language.
- The parser must also report any syntax errors.
- The parser should also recover from commonly occurring errors, so that it continue processing the remainder of its input

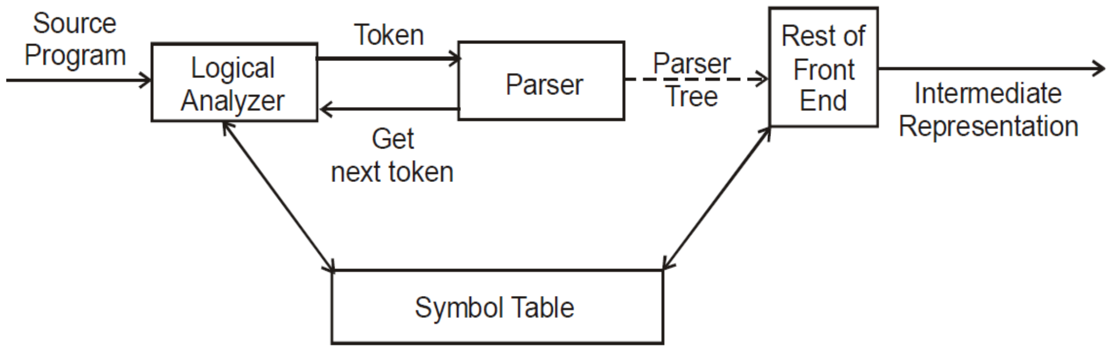


Figure 2.1

The methods commonly used in compilers are classified as:

- top down parsing or parsers
- bottom up parsing or parsers.

Top Down Parsers: Build parse trees starting from the root and work up to the leaves.

Bottom Up Parsers: Build the parse tree from the leaf work up to the root.

- In both the cases the input to the parser is scanned from left to right. One symbol at a time.
- The output of the parser is some representation of the parse free for the stream of tokens produced by the Lexical Analyzer.

The number of tasks that might be conducted during parsing are:

- Collecting information about various tokens into the symbol table.
- Performing type checking.
- Semantic analysis.
- Generating intermediate code.

But all these activities are dumped into the rest of front end box.

2.3 GRAMMAR

All the production rules together can be called as a grammar language defined by the grammer. A grammar derives string by beginning with the start symbol and repeatedly replacing a non-terminal by the right side of production for the non-terminal. These token strings that can be derived form the start symbol form the language difined by the grammar.

2.3.1 TYPES OF GRAMMAR

Type 0: Phrase Structured Grammer. They are grammar of the form shown below:

$\alpha \rightarrow \beta$ $\alpha , \beta \rightarrow$ strings

Type 1: Context Sensitve Grammar

$\alpha \rightarrow \beta$ $|\alpha \leq \beta|$ terminals are allowed in LHS

(Eg): $S \rightarrow aSBc$

$S \rightarrow abc$

$bB \rightarrow bb$

$bc \rightarrow bc$

Type 2. CFG (Context Free Grammar)

$\alpha \rightarrow \beta$ $|\alpha| \leq |\beta|$

α must be non terminal : no restriction for β

Type 3. Regular Grammar

$\alpha \rightarrow \beta$ $|\alpha| \leq |\beta|$ a Î V~

b $\rightarrow$ terminal followed by non-terminal

ab | a (eg) : $S \rightarrow as$

$S \rightarrow ab$

$B \rightarrow bc$

2.4 ERROR HANDLING

Good compiler should assist the programmer in identifying and locating errors. Programs can contain errors at many different levels.

Example: Errors can be:

- Lexical, such as misspelling an identifier, keyword or operator.
- Syntactic such as an arithmetic expressions with unbalanced parenthesis.
- Semantic such as an operator applied to an incompatible operand.
- Logical such as an infinitely recursive call.

Much of the error detection and recovery in a compiler is centered around the syntax analysis phase. 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 simple to state goals which are as follows:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect each subsequent errors.
- It should not significantly slow down the processing of correct programs.
- In difficult cases the error handler may have to guess what the programmer had in mind when the program was written.
- Errors may be detected when the error detector see a prefix of the input that is not a prefix of any string in the language.

Many of the errors could be classified simply

- 60% were punctuation errors
- 20% operator and operand errors
- 15% keyword errors and remaining 5% other kinds.

2.4.1 PUNCTUATION ERRORS

- Incorrect use of semicolons
- To use a comma in place of the semicolon etc.

Operator Error:

- to leave out the colon form ':='.

Misspellings of keywords:

- leaving out "*ln*" from write *ln*.

  Example: Error that is much more difficult to repair correctly is

- Missing of "begin" or "end".

How should an error handler report the presence of an error:

- It should report the place in the source program where an error is detected commonly.
- It is done by printing the offended line with a pointer to the position at which an error is detected.

- Sometimes informative, understandable diagnostic message is also included with the error message.

  Example: Semicolon missing at the position.

Once an error is detected how should the passer recover?

It is not good for a parser to quit after detecting the first error, because subsequent processing of the input may reveal additional errors.

So, there should be some form of error recovery by which the parser attempts to restore itself to a state where processing can continue.

- An inadequate job and recovery may introduce spurious errors.

Example: Syntactic error recovery may introduce spurious semantic errors that will later be detected by the semantic analysis or code generation phases.

Example: The variable '*i*' may be undefined, later when a statement with "*i*" is encountered, there may not be any syntactic mistakes in the statement but since there is no symbol table entry for '*i*', "*i* undefined" error is generated.

An error recovery strategy has to be carefully designed to take into account, the kinds of errors that are likely to occur and reasonable to process.

- Some compilers attempt error repair process by which the compiler attempts to guess what the programmer intended to write.

  Example: PL|C compiler

2.5 CONTEXT FREE GRAMMAR

Context Free Grammar specifies the syntax of a language. It is also called BNF (Backus Norm form).

A context free grammar has 4 components:

1. Set of tokens known as terminal symbols.
2. A set of non-terminals.
3. A set of productions where each production consists of a non-terminal on the left side of the production, an arrow, and a sequence of tokens and / or non-terminals called the right side of the productions.
4. One of the non-terminal is designated as the start symbol terminal.

   eg: Productions.

List	$\rightarrow$	List	+	Digit
List	$\rightarrow$	List	–	Digit
List	$\rightarrow$	Digit		
Digit	=	0 | 1 | 2 | ... | 9		

List, Digit are non-terminal. 0, 1, 2, ..., 9 are terminal.

The right sides of the three productions with non-terminal list on the left side can equivalently be grouped.

List $\rightarrow$ List + Digit | List - Digit | Digit.

Note : A single digit by itself is a list. If we take any list and follow it by a 't' or minus sign and then another digit we have a new a list.

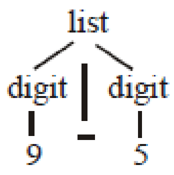
(E.g.) 9 is a list

9-5 is a list

Since 9 is a list which might be a digit and 5 is a digit.

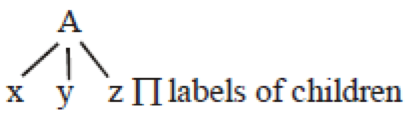
Parse Tree of 9-5:

- Each node in the tree is labeled be a grammar symbol.
- An interior node and its children correspond to a production.



Parse Trees for the production A $\rightarrow$ XYZ:

A parse tree may have an interior node labeled 'A' with 3 children labeled x,y and z.



Given a CFG, a parse tree is a tree with the following properties:

1. This root is labeled by start symbol.
2. Each leaf is labeled by a token or by *t* where *t* is an empty string.
3. Each interior node is labeled by a non-terminal.
4. If A is a non-terminal labeling some interior node and X_1, X_2 ....X_n are the lables of the

children of that node from left to right then A$\tilde{O}$$X_1$ X_2 X_3.........X_n is a productions.

Here X_1, X_2 ....X_n stands for a symbol ie either a terminal or a non-terminal.

If A $\rightarrow$ *t* then a node labeled '*a*' may have a single child labeled *t*.

NOTE : Any tree imparts a left to right order to its leaves based on the idea that if '*a*' and '*b*'are children with the same parent and '*a*' is the left of '*b*' then all descends of '*a*' are to the left of descendents of '*b*'.

Many programming language constructs have an inherently recursive structure that can be defined by context free grammar.

For Example: Statements such as:

"If E then S1 else S2 ", cannot be specified using the notation of regular expressions.

We can readily express the statement using the grammar production.

stmt $\rightarrow$ **if** expr **then** stmt **else** stmt

A context free grammar 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 consider grammars for programming languages.

(2) **Example:** if, then, else.

(3) Non-terminals are syntactic variables that denote sets of strings. They define the set of strings that can be generated by the grammar.

(4) **Example:** stmt, expr are non-terminals.

(5) 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.

(6) 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.

Grammar for Arithmetic Expression:

expr $\rightarrow$ expr op expr

expr $\rightarrow$ (expr)

expr $\rightarrow$ -expr

expr $\rightarrow$ id

op $\rightarrow$ +

op $\rightarrow$ $-$

op → *

op → /

op → ↑

In the grammar the terminal symbols are id, +, –, /, , ( )

Non-terminals are expr and op.

expr is Start symbol.

Notational Conventions:

(1) Terminals

(i) Lower case letters	→	a, b ... .
(ii) Operators	→	+ , – etc.
(iii) Punctuation Symbols	→	(, , etc.

(iv) The digits 0, 1, ..., 9.

(v) Bold face strings such as **id** or **if**.

(2) Non-Terminals

(i) Upper case letters A, B, C, ...

(ii) Letter S, which is usually the start symbol.

(iii) Lower case italic names *expr*, *stmt* ... .

(3) Upper case letters later in the alphabet such s X, Y, Z are grammar symbols, that is either Non-terminals or terminals.

(4) Lower case letters later in the alphabet such as *y*, *v* ... *r* represents strings of Terminals.

(5) Lower case Greek letters α, β, γ etc. represents strings of grammar symbols.

Thus a generic production could be written as $A \rightarrow \propto$ indicating that there is a single non-terminal A on the left of the arrow and a string of grammar symbols $\propto$ to the right of the arrow.

(6) If $A \rightarrow \alpha 1$, $A \rightarrow \alpha 2$ .... $A \rightarrow \alpha k$ are all productions called **A productions**.

We may also write $A \rightarrow \alpha 1|\alpha 2| \ldots |\alpha k$, where we call α1, α2 ... α*k* the alternatives for A.

(7) Unless otherwise stated the left side of the first production is the start symbol.

Using these short hand we could write the Grammar for expressions as

$E \rightarrow EAE \mid (E) \mid - E \mid id$

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

Derivations

The central idea of derivation is that a production is treated as a rewriting rule in which the non-terminal on the left is replaced by the strings on the right side of the production.

For Example: Consider the following grammar for arithmetic expression.

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

The production $E \rightarrow - E$ signifies that an expression preceded by a minus sign is also an expression.

So we can replace E by – E. We can describe this action of writing $E \Rightarrow - E$, which is read as "E drives – E".

We can take a single E and repeatedly apply productions in any order to obtain a sequence of replacements.

Example: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$.

We call such a sequence of replacements a derivation of (*id*) from E. This derivation provides a proof that one particular instance of an expression is the string (*id*).

If $\alpha_1 \Rightarrow \alpha_2 \ldots \alpha_n$ we say α_1 derives α_n and, $\alpha_1 \rightarrow \alpha_2$ is a production.

' $\Rightarrow$ ' means derives in one step.

$\overset{*}{\Rightarrow}$ " * " '0' or more steps.

$\overset{+}{\Rightarrow}$ "+" '1' on one step.

Given a Grammar 'G' with start symbol S we can use the $\overset{+}{\Rightarrow}$relation to define L(G), the language generated by 'G'.

(RMD) Rightmost, derivations are sometimes called **conical derivations**.

Note: If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$ then we say α is a left sentential form of the grammar G.

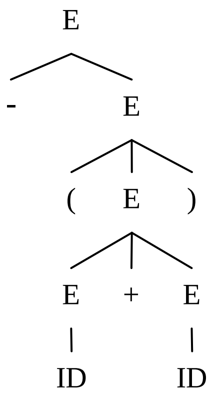
Parse Tree and Derivations

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacements order. (i.e., leftmost or rightmost information).

- Interior node is labeled by some non-terminal A.

- Children are labeled from left to right by the symbols in the right side of the production.
- The leaves of the parse tree are labeled by non-terminals or terminals and read from left to right.

  Example: parse tree for – (*id* + *id*) using LMD.



The sequence of parse trees constructed for the stated derivation is:

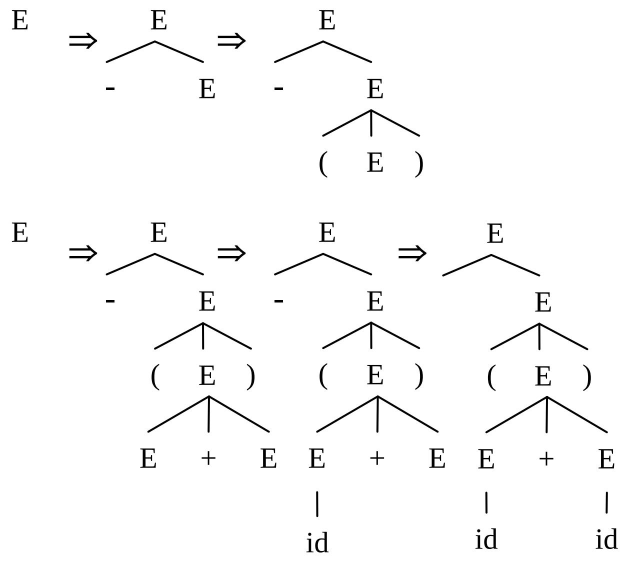


Figure 2.3

Example 1: Produce the string (*a* , (*a*, *a*)) with the following grammar, using LMD and RMD

S $\to$ *a* | $\wedge$ | (T)

T $\to$ T, S | S.

The terminals in the preceding grammar

$V_T = \{a, \wedge, (,), \}$

The non-terminals are:

$V_N = \{S, T\}$

Note: LMD $\Rightarrow$ Left Most Derivation

RMD $\Rightarrow$ Right Most Derivation.

RMD:

$S \rightarrow (T)$

$\rightarrow (T, S)$

$\rightarrow (T, (T))$

$\rightarrow (T, (T, S))$

$\rightarrow (T, (T, a))$

$\rightarrow (T, (S, a))$

$\rightarrow (T, (a, a))$

$\rightarrow (S, (a, a))$

$\rightarrow (a, (a, a))$.

LMD:

$S \rightarrow (T)$

$S \rightarrow (T, S)$

$S \rightarrow (S, S)$

$S \rightarrow (a, S)$

$\rightarrow (a, (T))$

$\rightarrow (a, (T, S))$

$\rightarrow (a, (S, S))$

$\rightarrow (a, (a, S))$

$\rightarrow (a, (a, a))$

Example 2: Produce the string $(((a, a), *, (a)), a)$ with the following grammar using LMD and RMD.

$$\boxed{\begin{array}{l} S \rightarrow a \,|*|\, (T) \\ T \rightarrow T, S \mid S \end{array}}$$

LMD

$S \rightarrow (T)$

$\rightarrow (T, S)$

$\rightarrow (S, S)$

$\rightarrow ((T), S)$

$\rightarrow ((T, S, S), S)$

$\rightarrow ((S, S, S), S)$

$\rightarrow (((T), S, S), S)$

RM D

$S \rightarrow (T)$

$\rightarrow (T, S)$

$\rightarrow (T, a)$

$\rightarrow ((T, S), a)$

$\rightarrow ((T, (T)), a)$

$\rightarrow ((T, (S)), a)$

$\rightarrow ((T, (a)), a)$

→ (((T, S), S, S), S)

→ (((S, S), S, S), S)

→ (((*a*, *a*), S, S), S)

→ (((*a*, *a*), *, (T)), S)

→ (((*a*, *a*, *, (S)), S)

→ (((*a*, *a*), *, (*a*)), S)

→ (((*a*, *a*), *, (*a*), S)

→ (((*a*, *a*), *, (*a*)), *a*)

→ ((T, S, (*a*)), *a*)

→ ((T, *, (*a*)), *a*)

→ ((S, *, (*a*)), *a*)

→ (((T), *, (*a*)), *a*)

→ (((T, S), *, (*a*)), *a*)

→ (((T, *a*), *, (*a*)), *a*)

→ (((S, *a*), *, (*a*)), *a*)

→ (((*a*, *a*), *, (*a*)), *a*).

Example 3: The sentence *id* + *id* is *id* has the two distinct left most derivation's which is shown below:

E ⇒ E + E	E ⇒ E * E
⇒ *id* + E	⇒ E + E * E
⇒ *id* + E * E	⇒ *id* + E * E
⇒ *id* + *id* * E	⇒ *id* + *id* * E
⇒ *id* + *id* * *id*	⇒ *id* + *id* * *id*.

The corresponding parse trees are shown below:

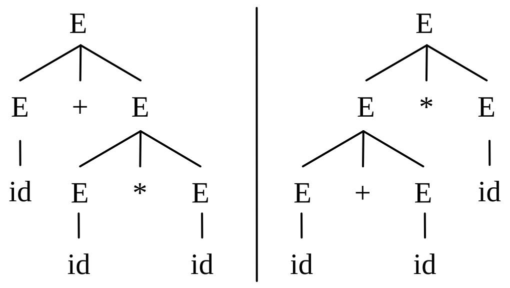
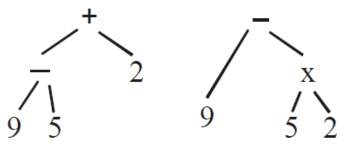


Figure 2.4

2.5.1 AMBIGUITY

A grammar can have more than one parse tree generating a given string of tokens such a grammar is said to be ambiguous.

String with more than one meaning had more than one parse tree.



(Eg) : (9-5)+2 or 9-(5+2) has 2 parse trees as

Associativity of operators:

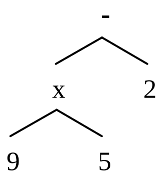
An operand with '*t*' sign on both sides of it is taken by the operator to its left.

9+5+2 is equivalent to (9+5)+2.

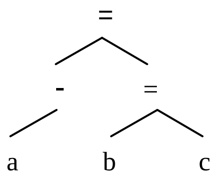
In most programming language the 4 arithmetic operator addition, subtraction, multiplication and division are left associative.

E.g.: a=b=c Here = is right associative

9 – 5 – 2 pare tree grows down towards left $\Rightarrow$



a = b = c grows down towards right $\Rightarrow$



Note: left associative are +, -, *, /

Precedence Operators:

We say that * has higher precedence than '+' if ' * ' takes its operends before 't' does.

For the productions,

factor $\rightarrow$ digit | (expr)

term $\rightarrow$ | term * factor

| factor

| term | factor

expr $\rightarrow$ expr + term

| expr - term

| term.

The resulting grammar is:

expr $\rightarrow$ expr + term | expr - term |term

term $\rightarrow$ term * factor | factor |term | factor

factor $\rightarrow$ digit | (expr)

2.6 WRITING A GRAMMAR : SOLUTION FOR THE PROBLEM OF AMBIGUOUS GRAMMAR

We can use ambiguous grammar together with disambiguating rules that "throw away" undesirable parse trees and leave only one tree for each sentence.

Regular Expression* (vs) *Context Free Grammar

Every construct that can be described by a regular expression can also be described by a grammar.

For example the grammar for the regular expression $(a|b)^*abb$ is:

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \in$

NFA of $(a|b)^*abb$ is

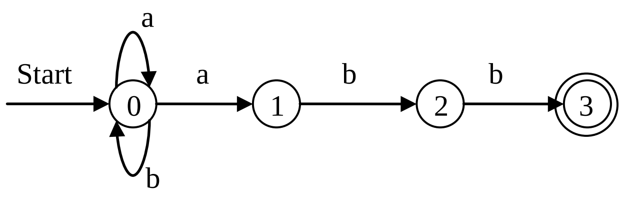


Figure 2.5

We can easily convert a NFA into a grammar that generates the same language as recognized by the NFA.

The grammar above was constructed from the NFA using the following construction:

- For each state; 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 '*i*' is an accepting state introduce

  $A_i \rightarrow E$

- If '*i* is the start state make A_i be the start symbol of the grammar.

Why do we use regular expression to define the lexical syntax of a language?

(1) The lexical rules of a language are frequently quite simple and to describe them we do not need a notation as powerful as grammars.

(2) Regular expressions generally provide a more concise and easier to understand notation for tokens in the grammars.

(3) More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

(4) Separating the syntactic structureof a language into lexical and non-lexicalpart provide a convenient way of modularizing the front end of a compiler into two manageable sized components.

- Regular expressions are most useful for describing the structure of lexical constructs such as identifiers, constants, etc.
- Grammars are most useful in describing nested structures such as balanced parenthesis, and statements like "if then else", etc.

2.6.1 ELIMINATING AMBIGUITY

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.

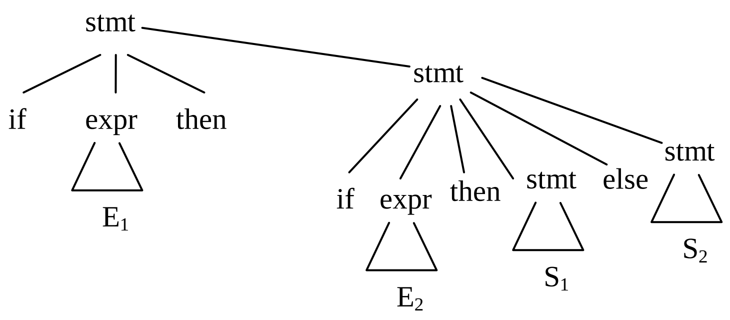
For example we shall eliminate the ambiguity, from the following dangling else grammar.

stmt $\rightarrow$ **if** expr **then** stmt

| **if** expr **then** stmt **else** stmt

| other

The grammar is ambiguous, since it has two parse trees as shown below.



and

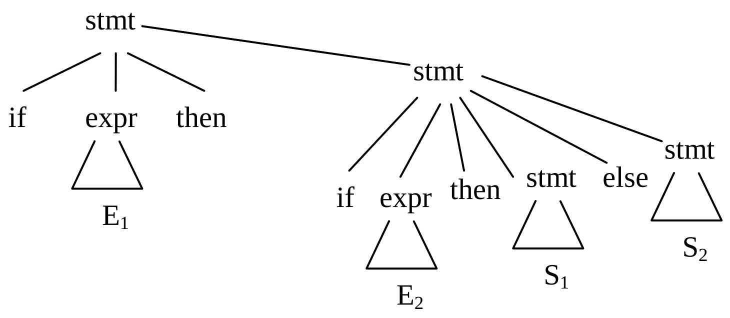


Figure 3.6

In all the programming languages with conditional statements of this form, the first parse tree is preferred.

2.6.2 DISAMBIGUATING RULE

"Match each else with the closest previous unmatched then".

The idea is that a statement appearing between a then and an else must be matched.

A matched statement is either an if then else statement containing no unmatched statement ot it is any other kind of unconditional statement.

stmt	$\rightarrow$	matched.stmt
	|	unmatched.stmt
Matched-stmt	$\rightarrow$	if expr then matched-start else
	|	other matched-stmt
Unmatched stmt	$\rightarrow$	if expr then stmt

| if expr then matched-stmt else unmatched-stmt

2.6.3 ELIMINATION OF LEFT RECURSION

A grammar is left recursive if it has a non-terminal A such that there is derivation A $\overset{*}{\Rightarrow}$ Aα for some string α.

Top down passing methods cannot handle left recursive grammars so a transformation that eliminates left recursion is needed.

The left recursive pair of productions $A \rightarrow A\alpha \mid \beta$ could be replaced by the non-left recursive productions.

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha\, A' | \in$$

without changing the set of strings derivable from A.

Example: Consider the following grammar for arithmetic expressions:

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

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

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

Eliminating the left recursion of E and T we obtain:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

No matter how many A productions there are we can eliminate immediate left recursion from then by the following technique.

(1) Group the productions as:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \ldots A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \beta_n.$$

where no β_ι begins with an A. Then we replace the A productions by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \ldots \alpha_m A' \mid \in$$

The non-terminal A generates the same strings as before but is no longer left recursive.

This procedure eliminates all immediate left recursion but it does not eliminate left recursion involving derivations of two or more steps.

Example: Consider the grammar,

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \in .$$

The non-terminal S is left recursive because:

$$S \rightarrow Aa \rightarrow Sda.$$

NOTE: A grammar is said to have no cycles if the derivations are of the form $A \overset{+}{\Rightarrow} A$.

Algorithm to eliminate left recursion (due to derivations)

Input: Grammar G with no cycles or $\in$ productions.

Output: An equivalent Grammar with no left recursion.

Method: Apply the alg. to G. The resulting non-left recursive grammar may have $\in$ productions.

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

2.6.4 LEFT FACTORING

Example: If we have the two productions,

stmt $\rightarrow$ *if* expr *then* stmt *else* stmt

| *if* expr *then* stmt

On seeing the input token if we cannot immediately tell which production to choose to expand “stmt”.

In general if $A \rightarrow \alpha\beta_1|\alpha\beta_2$ are two A productions and the input begins with a non-empty string desired from α, we do not know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$. However we may defer the decision by expanding A to A’. Then after seeing the input derived from α we expand A’ to β_1 or to β_2.

That is, left factored the original productions become:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1|\beta_2.$$

Algorithm: Left factoring a Grammar

Input: Grammar G

Output: An equivalent left factored grammar.

Method:

- For each non-terminal A find the longest prefix α common to two or more of its alternatives.
- If $\alpha \neq \in$ replace all the A productions

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \gamma$ where γ represents all alternatives that do not begin with α by

  $$A \rightarrow \alpha A' \mid \gamma$$

  $$A' \rightarrow \beta_1 \mid \beta_2 \ldots \beta_n.$$

  Here A′ is a new non-terminal.
- Repeatedly apply the transformation until no two alternatives for a non-terminal have a common prefix.

The following grammar abstracts the dangling else problem.

$$S \rightarrow i\text{EtS} \mid i\text{E}t\text{S}e\text{S} \mid a$$

$$E \rightarrow b$$

E and S are for expression and stmt *i, t, e* stands for if then and else.

Left factored grammar becomes

$$S \rightarrow i\text{EtSS}' \mid a$$

$$S' \rightarrow e\text{S} \mid \in$$

$$E \rightarrow b$$

Thus we may expand S to iEtSS′ on input i and wait until iEtS has been seen to decide whether to expand S′ to *es* or to $\in$.

2.7 TOP DOWN PARSING

Recursive Descent Parsing

Top down parsing can be viewed as an attempt to find a leftmost derivation for an input string:

It may involve back tracking that is making repeated scans of the input.

It is an attempt to construct a parse tree from the root and creating the nodes of the tree in pre-order.

Disadvantage of Backtracking

- Not very efficient.

Note: We have to keep track of the input when backtracking takes place.

Example: Let the grammar be:

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

To construct a parse tree for the string $w = cad$, we initially create a tree consisting of a single node labelled S. We shall consider the first production and obtain the tree as follows,

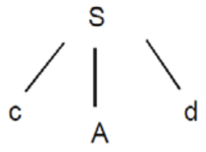


Figure 2.7

The leftmost leaf labeled *c* matches the first symbol of ω, so we advance the input pointer to 'a', (the second symbol of *w*), and consider the next leaf labeled A.

- We then expand A, using the first alternative for A to obtain the following tree

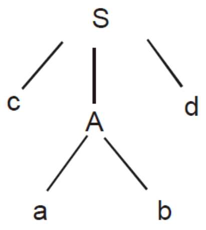


Figure 2.8

- *b* does not match with the third input symbol '*d*' we see whether there is another alternative for A.
- In going back to A we must reset the input pointer to position 2.

We now try the second alternative for A to obtain the following tree:

Since we have produced a parse tree for 'ω' we halt and announce successful completion of parsing.

Recursive descent parser, cannot handle left recursive grammars. So, a transformation that eliminates left recursion is needed.

A left recursive grammar can cause a recursive descent parser to go into an infinite loop.

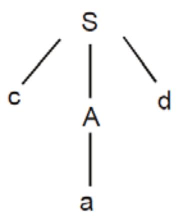


Figure 2.9

(since A → a production rule is applied now)

2.7.1 PREDICTIVE PARSERS

Recursive descent parser that needs no back-tracking is called a predictive parser. This can be accomplished by carefully writing a grammar eliminating left recursion from it and left factoring the resulting grammar to obtain a grammar that can be parsed by a predictive parser.

In the production $A \rightarrow \alpha 1 \mid \alpha 2 \ldots \mid \alpha n$ the proper alternative must be detectable by looking at only the first symbol it derives.

For Example: If we have the productions.

statement → **if** expr **then** stmt **else** stmt

| **while** expr **do** stmt

| **begin** stmt list **end**

Then the keyword if, while and begin tell us which alternative is the only one that would possibly succeed if we're to find a statement.

2.8 GENERAL STRATEGIES RECURSIVE DESCENT PARSER PREDICTIVE PARSER

2.8.1 TRANSITION DIAGRAMS FOR PREDICTIVE PARSER

We can create a transition diagram as a plan for a predictive parser.

To construct the transition diagram of a predictive parser from a grammar, first eliminate left recursion from the grammar then left factor the grammar. Then for each non-terminal A do the following:

(1) Create an initial and final state.

(2) For each production $A \rightarrow X_1X_2 \ldots X_n$.

Create a path from the initial to the final state with edges labeled $X_1X_2\ldots X_n$.

Example: Consider the following grammar:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id \mid (E)$

The collection of transition diagrams for the preceding grammar are:

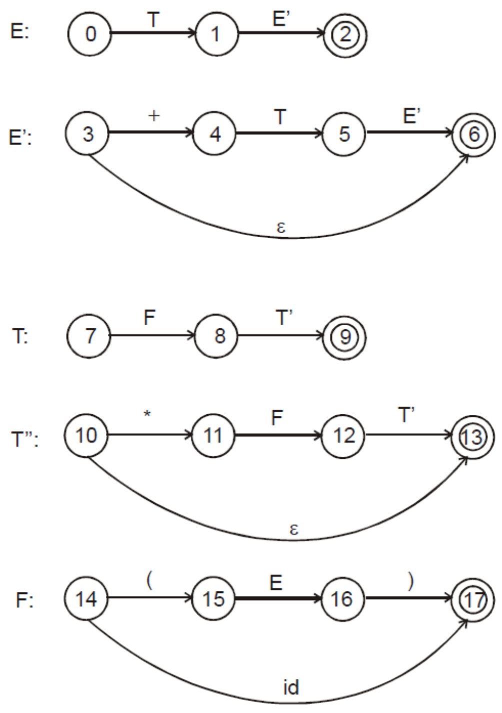
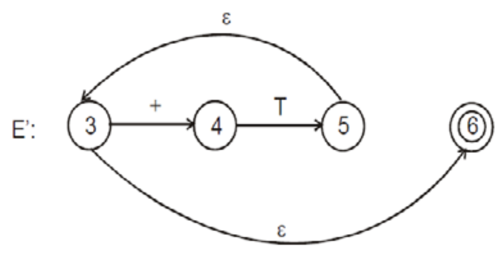
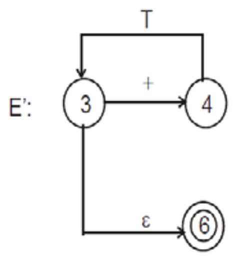
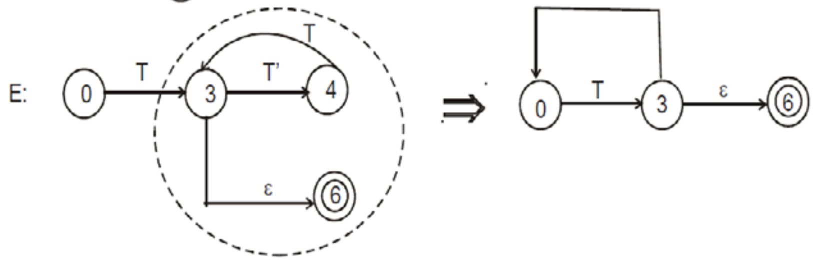


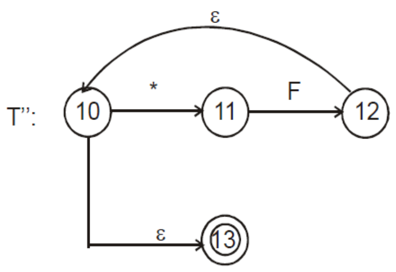
Figure 2.10

On Simplification





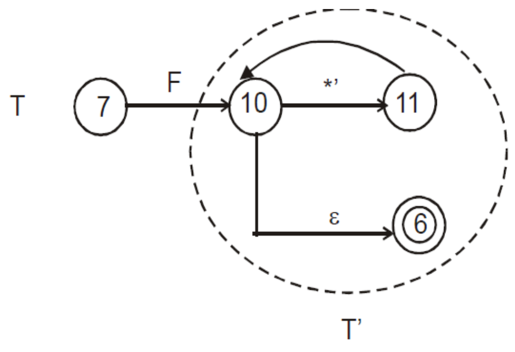




String generated

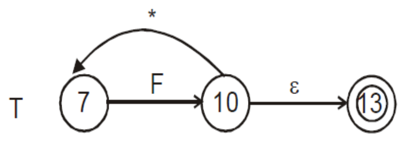
F * F * F ...

Figure 2.11



String generated

F * F * F ...



String generated

F * F * F ...

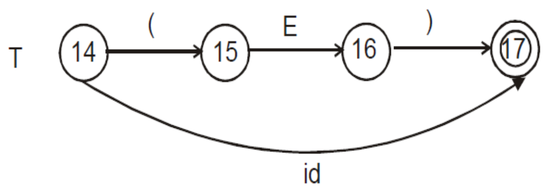


Figure 2.12

We must eliminate non-determinism if we, want to build a predictive parser.

Non-determinism, means there is more than one transition from a state on the same input. If ambiguity occurs we should resolve it.

If non-determinism cannot be eliminated we cannot build a predictive parser, but we could build a recursive descent parser using backtracking to systematically try all possibilities.

The complete set of diagram after simplification are:

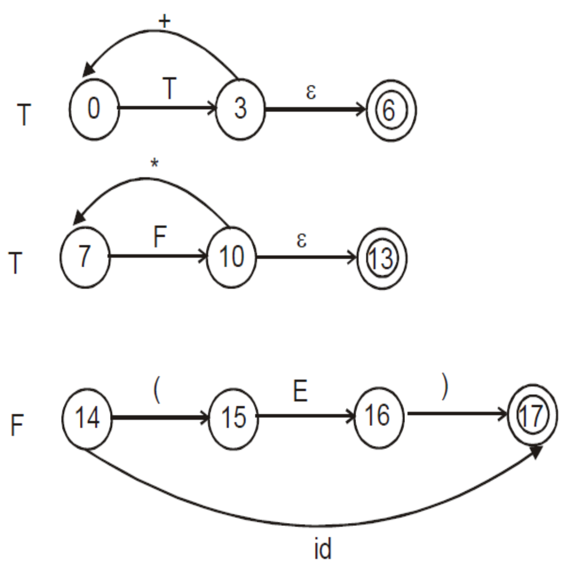


Figure 2.13

A ‘C’ implementation of the predictive parser based on the preceding simplified transition diagram runs, 20 – 25% faster than the ‘C’ implementation of predictive parser based on the transition.

2.8.2 NON-RECURSIVE PREDICTIVE PARSING

It is possible to build a non-recursive predictive parser by maintaining a stack. This parser looks up the production to be applied in a parsing table.

A table driven predictive parser has an:

- input buffer
- a stack
- parsing table
- output stream.
- the input buffer contains the string to be passed followed by $ which 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 \$.
- 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 parser is controlled by a program that behaves as follows:

The program compares 'X', the symbol on top of the stack and 'a' the current input symbol.

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 non-terminal the program consults entry M[X, *a*] of the parsing table M.

If for example, $M[X, a] = \{X \rightarrow UVW\}$ the parser replaces X on top of the stack by WVU.

As an output we should assume that the parser just prints the production used.

If M[X, *a*] = error the parser calls an error recovery routine.

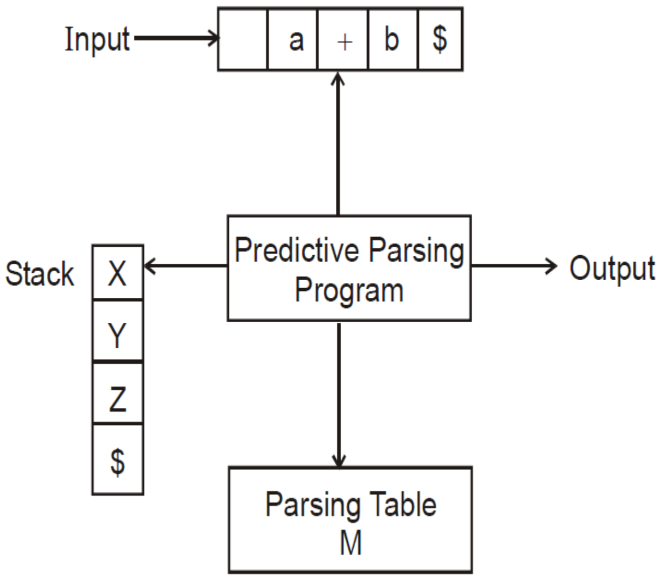


Figure 2.14: Model of non-recursive predictive parser

Algorithm:

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

Output: If ω is in L(G) a leftmost derivation of 'ω' otherwise an error indication.

Method:

Initially the parser is in a configuration in which it has \$S on the stack with 'S' the start symbol of G on top and ω\$ in the input buffer.

This program utilizes the predictive parsing table M to produce a parse for the input.

```
Set input to point to the first symbol of ω$
Repeat
Let X be the top stack symbol and a the symbol pointed to by input.
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance input
        else error ( )
    else
        if M[X, a] = X → Y1 , Y2 ... YK then begin
            pop X from the stack;
        push yk, yk-1 … y1 onto the stack with y1 on top;
        output the production X → y1, y2, ..., yk
      end
    else
        error ()
    until X = $
```

Example: Consider the grammar:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

With input *id + id * id* the predictive parser makes the sequence of moves as shown below. This parser traces out a leftmost derivation for the input, the output productions are those of a leftmost derivation.

Step 1: FIRST AND FOLLOW

The functions FIRST and FOLLOW allow us to fill in the entries of a predictive parsing table.

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

1) If X is terminal then FIRST (X) is {X}.

   (i.e) First (terminal) = terminal.

2) If $X \rightarrow \varepsilon$ is a production then add ε to FIRST (X).

3) first (non-terminal) = first (terminal)

   terminal= 1st terminal symbol produced by the series of non-terminal.

To Compute FOLLOW ($) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set. The NT to which Follow is computed should not be in the both sides of the production.

1) If S is the starting symbol

   FOLLOW (S) = {$}.

2) If there is a production $A \rightarrow \alpha B\beta$ is a rule then

   FOLLOW (B) = FIRST (β) except 'ε'.

3) If there is a production $A \rightarrow \alpha B$

   (or) If $A \rightarrow \alpha B\beta$; $\beta \rightarrow \varepsilon$

   $\boxed{\text{Follow (B) = FOLLOW (A).}}$

Considering the following grammar let us compute the FIRST and FOLLOW

$E \rightarrow TE'$ ... (1)

$E' \rightarrow +TE'|\varepsilon$ ... (2)

$T \rightarrow FT'$ ... (3)

$T' \rightarrow *FT'|\varepsilon$ ... (4)

$F \rightarrow (E)|id$ ... (5)

FIRST (E) = {(, id}

FIRST (E′) = {+, ε}

FIRST (T) = {(, id}

[**Explanation:** First(E) = T by (1) which is a non-terminal

Hence First(E) = First(T)

First(E) = F which is again a NT by (3)

Hence First(E) = First (F)

FIRST (T′) = {*, ε}

FIRST (F) = {(, id}

= (, id (by 5)]

NOTE:

- In the right hand side of the production fix the element for which the follow is to be calculated as 'B', the left side elements of B is 'α' and the elements in the right side of 'B' is 'β'.
- Then try to apply all possible rules (1), (2) and (3) for the follow calculation.

1. **FOLLOW (E)**

   Find the production in which E is in the right side and more over the left side should not have the same NT.

   $$\begin{array}{lccc} \text{Consider } F \rightarrow & (& E &) \\ & \downarrow & \downarrow & \downarrow \\ A \rightarrow & \alpha & B & \beta \end{array}$$

   FOLLOW (E) = FIRST (β) by rule (2)

   = {) }

   As 'E' is the start symbol add $ to the follow calculation by rule (1).

   NOTE: FOLLOW (E) = {), $}

   Rule (3) is not applicable.

2. **FOLLOW (E1)**

   $$\begin{array}{lcccl} \text{Consider } E \rightarrow & T & E' & & \\ & \downarrow & \downarrow & \downarrow & \beta = \varepsilon \\ A \rightarrow & \alpha & B & \beta & \end{array}$$

   FOLLOW (E′) = FOLLOW (E) by rule (3).

   Since β in ε, Follow (E′) = Follow (E).

   NOTE: Role (1) and (2) are not applicable

   ∴ FOLLOW (E1) = {) , $ }

3. **FOLLOW (T)**

   Consider E′ → + TE′

   FOLLOW (T) = FIRST (E′) except ε by rule (2)

$= \{+\}.$

$$\begin{array}{lccc} \text{Consider } E' \rightarrow & + & T & E' \text{ where } E' \Rightarrow \varepsilon. \\ & \downarrow & \downarrow & \downarrow \\ A \rightarrow & \alpha & B & \beta \end{array}$$

FOLLOW (T) = FOLLOW (E′) by rule (3)

= {) , $}

∴ FOLLOW (T) = { + ,) , $}

Note: Rule (1) not applicable.

Another way to compute FOLLOW (T) using the production

$E \rightarrow TE'$

$E \rightarrow TE'$

$A \rightarrow \alpha\, B\, \beta$

FOLLOW (T) = FIRST (E′) except ε by rule (2)

= {+ }

Consider $E \rightarrow TE'$ where $E' \rightarrow \varepsilon$

$A \rightarrow \alpha\, B\, \beta$

FOLLOW (T) = FOLLOW (E) by rule (3)

= {) , $}

∴ FOLLOW (T) { + ,) , $}

NOTE: Rule (1) is not applicable as ‘T’ is not the start symbol.

4. FOLLOW (T′)

$$\begin{array}{lccc} \text{Consider } T \rightarrow & F & T' & \varepsilon \\ & \downarrow & \downarrow & \downarrow \\ A \rightarrow & \alpha & B & \beta \end{array}$$

∴ FOLLOW (T′) = FOLLOW (T) by rule (3) Rules (1) and (2) are not applicable

∴ FOLLOW (T′) = { + ,) , $}

5. FOLLOW (F)

$$\begin{array}{lccc} \text{Consider } T' \rightarrow & \alpha & F & T' \\ & \downarrow & \downarrow & \downarrow \\ A \rightarrow & \alpha & B & \beta \end{array}$$

FOLLOW (F) = FIRST (T′) except ε by rule (2)

= {*}

Consider $T' \rightarrow * FT'$ where $T' \rightarrow \varepsilon$

$A \rightarrow \alpha B \beta$

FOLLOW (F) = FOLLOW (T′) by rule (3)

∴ FOLLOW (F) = { * , + ,) , $}

Another way to compute FOLLOW (F) using the production:

$$\begin{array}{lccc} T \rightarrow & & F & T' \\ & \downarrow & \downarrow & \downarrow \\ A \rightarrow & \alpha & B & \beta \end{array}$$

FOLLOW (F) = FIRST (T′) except ε by rule (2)

= {*}

Consider $T \rightarrow FT'$ where $T' \rightarrow \varepsilon$

FOLLOW (F) = FOLLOW (T) by rule (3)

= {+ ,) , $ }

∴ FOLLOW (F) = FIRST (T′) + FOLLOW (T)

= {* , + ,) , $}

The following algorithm can be used to construct predictive passing table.

Algorithm

Grammar G

Output: Parsing table M

Note: Input:

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (\varepsilon) | id$

Method:

1) For each production $A \rightarrow \alpha$ of the grammar do step 2 and 3.
2) For each terminal a in First (α) add $A \rightarrow \alpha$ to M[A, *a*].
3) If ε is in FIRST (α) add $A \rightarrow \varepsilon$ to M[A, *b*]

   for each terminal *b* in FOLLOW (A)

   If ε is in FIRST (α) and $ is in FOLLOW (A)

   add $A \rightarrow \alpha$ to M[A, $].

   NOTE: In this case 'α' must be ε.
4) Make each undefined entry of M be error.

NOTE: The parsing table produced by the preceding algorithm is shown as below.

Step 2: Parsing

Construct of Parsing Table.

$E \rightarrow TE'$

$E' \rightarrow + TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

Construct the parsing table.

Rows $\rightarrow$ Non-Terminals.

Columns $\rightarrow$ Terminals.

Non-Terminal	**Input Symbol**					
	Id	+	*	(	)	S
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)		

Procedure to fill the Predictive Parse Table

Consider the I production (Revised)

$E \rightarrow TE'$

Consider First (E) = { (, id }

Under the Row E, column ‘(’ write the production $E \rightarrow TE'$

Under the Row E, column id write the production $E \rightarrow TE'$.

Consider the II Production

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

First (E′) = {+, ε}

Row E′, Column + write the production $E' \rightarrow + TE'$.

Since First (E′) has ε consider the Follow (E′).

Follow (E′) = {) , $ }

Having Row as E′ and Column as) write the production $E' \rightarrow \varepsilon$.

Row as E′ and Column as $ write the production $E' \rightarrow \varepsilon$.

Consider III Production

$T \rightarrow FT'$

First (T) = { (, id }

Row-T, Column ‘(’ write the production $T \rightarrow FT'$

Row-T, Column id write the production $T \rightarrow FT'$.

Consider IV production

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

First (T′) = { * , ε }

Row T′, Column * write the production T′ → *FT′ | ε

Since ε is in First (T′) use the Follow (T′) Follow (T′) = { + ,) , $ }

Row T′ Column + write production T′ → ε

Row T′ Column) write production T′ → ε

Row T′ Column $ write production T′ → ε

Consider V Production

F → (E) | id

First (F) = { (, id }

Row F Column C write the production F → (E)

Row F Column id write the production F → (E) | id

For the remaining, same terminal pop in out.

(e.g.,) (+, +) pop

(id, id) pop.

NOTE: If FIRST (X) has ‘ε’ then find follow (X) and include X → ε in M[X, *b*], where *b* is a terminal in FOLLOW (X) LL(1) grammars.

The preceding Algorithm can be applied to any grammar G to produce a parsing table.

For some grammars lower the table may have some entries that are multiply defined. This occurs when the grammar is ambiguous.

Conclusion: The string id + id * id is in accordance to the grammar.

Step 3: Do the parsing

Stack	Input	Output
\$E′	id + id *id\$	Look for (E, id) in table of step2, it results in
\$E′T	id + id *id\$	$E \to TE'$ [POP E, push RHS]
\$E′T′F	id + id *id\$	$T \to FT'$ of the production]
\$E′T′	id + id *id\$	$F \to id$
\$E′T′	+ id *id\$	pop (.id, id) and advance
\$E′	+ id *id\$	$T' \to \varepsilon$
\$E′T+′	+ id *id\$	$E' \to +TE'$
\$E′T	id *id\$	
\$E′T′F	id *id\$	$T \to *FT'$
\$E′T′id	id *id\$	$F \to id$
\$E′T′	*id\$	
\$E′T′F*	*id\$	$T' \to *FT'$
\$E′T′F	id\$	
\$E′T′id	id\$	$E \to id$
\$E′T′	\$	
\$E′	\$	$T' \to \varepsilon$
\$	\$	$E' \to \varepsilon$

Hence the string is accepted.

Example 2: Consider the following grammar:

$S \to iEtSS'|a$

$S' \to eS|\varepsilon$

$E \to b$

First (S) $= \{i, a\}$

First (S′) = {*e*, ε}

First (E) = {*b*}

Follow (S) = First (S’)

= {*e*, $} → as ‘S’ is the start symbol

FOLLOW (S′) = Follow (S) = {*e*, $}

Consider S ⇒ iEt SS′ ⇒ FOLLOW (E) = FIRST (β) *ds* β → *t*SS′ FIRST (β) = {*t*}.

∴ FOLLOW (E) = First (*t*SS′) = {*t*} $.

The parsing table for this grammar is shown below:

Non-terminal	**NT**	**Input symbol**					
		a	**b**	**e**	**i**	**t**	**s**
FIRST (S) = {*i*, *a*}	S	S → *a*			S → *i*E*t*SS′		
FOLLOW (S′) = FOLLOW (S) = {*e*, $}	S′			S′ → ε S′ → *e*S			S′ → ε
FIRST (E) = {*b*}	E		E → *b*				

The entry for M[S′, *e*] contains both S′ → *es* and S′ → ε

Thus the grammar is ambiguous.

We can resolve the ambiguity if we choose S′ → *e*S among S′ → ε and S′ → *e*S. This choice corresponds to associating else with the closest previous then

S′ → E is surely wrong as it will satisfies the production use of an unmatched statement.

(i.e) *dr*. S → *i*E*t*SS′, if S′ → E the production best

S → *i*E*t*S which is III to:

unmatched statement → if expr then statement.

Definition: 1

A grammar whose parsing table has no multiply defined entries is said to be LL(1).

1st L	→	scanning the input from left to right
2nd L	→	for producing a left most derivation

1	$\rightarrow$	for using one input symbol of look ahead at each step to make parsing action decisions.

No ambiguous or left recursive grammar can be LL(1).

It can also be shown that *x* grammar *r* is LL(1) if any only if whenever $\boxed{A \rightarrow \alpha|\beta}$ are two distinct productions of G, the following conditions holds.

1) For no terminal '*a*' do both α and β derive strings beginning with '*a*'.
2) Atmost one of α and β can derive the empty string.
3) If $\beta \overset{*}{\Rightarrow} \varepsilon$ then 'α' does not derive any string beginning with a terminal in FOLLOW (A)

Example of LL(1) Grammar	**Example of a Grammar which is not LL(1)**
$E \rightarrow TE'$ $E' \rightarrow +TE'\|\varepsilon$ $T \rightarrow FT'$ $T' \rightarrow *FT'\|\varepsilon$ $F \rightarrow (E)\|id$ By Rule (3), consider $T \rightarrow *FT'/\varepsilon$ FOLLOW (T′) = {+,), $} α is * FT′ and β is ε. 'α' does not starts with +,), or $ Thus rule 3 is satisfied By (2), β is ε and 'α' is not ε By (1), 'α' string begins with '+' and β does not derive strings beginning with '+'.	$S \rightarrow iEts\|iEtSeS\|a$ $E \rightarrow b$ (1) Rule is not satisfied. Consider $S \rightarrow iEts\|iEtSeS$. Here both α and β are beginning with (.) Thus rule (1) is not satisfied.

Disadvantages of Predictive Parsing

1) It is difficult in writing a grammar for the source language such that a predictive passer can be constructed from the grammar.
2) Left recursion elimination and left factoring make the resulting grammar hard to read and difficult to use for translation purposes.

Error Recovery in Predictive Parsing

Two Possible Reasons for Error

1) An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol.
2) When non-terminal A is on top of the stack '*a*' is the next input symbol and the parsing table entry M[A, *a*] is empty.

Panic mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens (delimiters **eg.** ';" "end") appears.

The parsing table seen earlier is modified by adding synchronizing in the following set of each non-terminal.

Non-Terminal	**Input symbol**					
	id	+	*	**(**	**)**	**S**
follow (E) = {), $}	E→TE'			E→TE'	Sync	Sync
follow (E') = {), $}		**E'**→+TE'			E'→ε	E'→ε
follow (T) = {+,), $}	T→FT'	**Sync**		T→FT'	Sync	Sync
follow (T′) = {+,), $}		**T'**→ε	T'→FT		T'→ε	T'→ε
follow (F) = {+, *,), $}	F→id	**Sync**	Sync	F'→(E)	Sync	Sync

NOTE: The same table discussed earlier is modified by adding sync in M(X, *c*) where *c*′ is a terminal in Follow (X). If the respective cells are already filled, leave it undisturbed.

- If the parser looks up entry M[A, *a*] and finds that it is blank, then the input symbol '*a*' is skipped.
- If the entry is sync then the non-terminal on top of the stack is popped in an attempt to resume parsing.
- If a token on top of the stack does not match the input symbol, then we POP the token from the stack.

On the erroneous input + id * + id the parser and the error recovery mechanism behaves as follows:

Stack	***Input***	***Remark***	***Stack***	***Input***	***Remark***
$E	+id*+id$	error, skip "+" ∵M[E,+]=empty	$E′T′F	+id$	error M[F, +] = synch F has been popped

$\$E$	$id*+id\$$	id is in First (E)	$\$E'T'$	$+id\$$	
$\$E'T$	$id*+id\$$		$\$E'$ $\because \$E' \rightarrow TE'$	$+id\$$	
$\$E'T'F$ $\because T \rightarrow FT'$	$id*+id\$$		$\$E'T'+$ $\because E \rightarrow +TE'$	$+id\$$	
$\$E'T'id$	$id*+id\$$		$\$E'T$	$+id\$$	
$\$E'T'$	$*+id\$$		$\$E'T'F$ $\because T \rightarrow FT'$	$id\$$	
$\$E'T'F*$ $\because T' \rightarrow *FT'$	$*+id\$$		$\$E'T'id$ $\because F \rightarrow id$ $\$E'T'$ $\$E' \because T' \rightarrow E$ $\$ \because E' \rightarrow E$	$id\$$ $\$$ $\$$ $\$$	

The above discussion of panic mode recovery does not address the important issue of error messages. In general informative error messages have to be supplied by the compiler designer.

Phrase Level Recovery:

It is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change insert or delete symbols on the input and issue appropriate error messages.

2.9 STACK IMPLEMENTATION OF SHIFT REDUCE PARSER

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 ‘ω’ to be parsed. ‘$’ may be used to mark the bottom of the stack and also the right end of the input.

Initially,	**Stack**	**Input Buffer**
	$\$$	$i + i * i\$$

- The parser shifts zero or more input symbols onto the stack until *a* handle β is on top of the stack.

$S \rightarrow aAcBe$

$A \rightarrow Ab|b$

$B \rightarrow d$ $\omega = abbcde$

Stack	Input	Action
$a	bbcde$	shift
$a	bbcde$	shift
$ab	bcde$	reduce A $\rightarrow$ b
$aA	bcde$	shift
$aAb	cde$	reduce A $\rightarrow$ Ab
$aA	cde$	shift
$aAc	de$	shift
$aAcd	e$	reduce B $\rightarrow$ d
$aAcB	e$	shift
$aAcBe	$	reduce
$S	$	accept

- The parser then reduces β to the left side of the appropriate production.
- The parser repeats the cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

Stack	Input	Action
$	i + i * i $	shift
$i	+ i * i $	reduce by E $\rightarrow$ i
$E	+ i * i $	shift
$E +	i * i $	shift
$E + i	* i $	reduce by E $\rightarrow$ i
$E + E	* i $	shift
$E + E *	i $	shift
$E + E * i	$	reduce by E $\rightarrow$ i

$E + E * E	$	reduce by E $\rightarrow$ E * E
$E + E	$	reduce by E $\rightarrow$ E + E
$E	$	accept

Four possible actions:

1) **Shift:** Next input symbol is shifted onto the top of the stack.
2) **Reduce:** The parser reduces the right end of the handle at the top of the stack to the left end of the handle, by deciding with what non-terminal to replace the handle.
3) **Accept:** The parser announces successful completion of parsing.
4) In an error action, the parser discovers that syntax error has occurred and calls an error recovery routine.

Viable Prefixes:

The set of prefixes of right sentenial forms that can appear on the stack of a shift reduce parses are called viable prefixes.

Conflicts during Shift Reduce Parsing:

There are context free grammars for which shift reduce passing cannot be used. Such grammars can reach a configuration in which the parser knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (shift-reduce conflict) or cannot decide which of the several reductions to make (reduce-reduce conflict).

These grammars are referred to as non-LR grammars.

An ambiguous grammar can never be LR.

Example:

stmt $\rightarrow$ if expr then stmt

|if expr then stmt else stmt

|other

$\therefore$ **STACK**	**INPUT**
if expr then stmt	else .... $.

We cannot tell whether if expr then statement is the handle.

Here, there is a shift-reduce conflict as we cannot determine, whether it is correct to reduce

“if expr then statement” to “statement” or it might be correct to shift ‘else’ and then to look for another statement to complete the alternate “if expr then statement else statement”.

Thus, we cannot tell whether to shift or reduce in this case, so the grammar is not LR(1).

L → left to right scanning of input

R → constructing a rightmost derivative in reverse

1 → number of input symbols of lookahead that are used in making parsing decisions.

Example: of reduce-reduce conflict.

Suppose the lexical analyser returns token 'id' for all identifier regardless of usage and the grammar might therefore have productions such as:

1) statement	→	id (parameter_list)
2) statement	→	expr := epr
3) parameter_list	→	parameter_list, parameter
4) parameter_list	→	parameter
5) parameter	→	id
6) expr	→	id (expr_list)
7) expr	→	id
8) expr_list	→	expr_list, expr
9) expr_list	→	expr

The statement A(I, J) wouldappear as the token stream id (id, id) to the parser.

After shifting the first 3 tokens onto the stack, a shift reduce parser would be in configuration.

STACK	**INPUT**
id (id	, id) ... .

It is clear that id on the top of stack is to be reduced but by which production?

We would choose 7th production rule to reduce "id" to expr.

ln this method, shift reduce parsing can utilize information far down in the stack to guide the parse.

The id on top of the stack must be reduced but by which production? It should be (5) if A is a procedure and (7) if A is an array.

Solution:

Change the token ‘id’ in production (1) to ‘procid’ and to use a lexical analyser that returns token “procid” when it recognises an identifier which is the name of a procedure, by consulting the symbol table before returning a token.

If this modification is made, then on processing A (I, J), the passer would be in the configuration.

STACK	**INPUT**
.. procid (id	, id) ....

In this case we choose the 5th production rule for reducing the top of the stack symbol “id” to parameter.

OPERATOR PRECEDENCE PARSING

The two conditions should be satisfied for OPP (Operator Precedence Parser)

1) no production on right side is ε
2) no production on right side has two adjacent non-terminals

$$\left.\begin{array}{l}\text{Ex: } E \rightarrow EAE \mid (E) \mid - E \mid id \\ A \rightarrow + \mid - \mid * \mid / \mid \uparrow\end{array}\right\} \text{ is not operator grammar.}$$

However if we substitute for A we get operator grammar:

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

In OPP we find 3 disjoint precedence relations <, =, > between pair of terminals. They select the handles:

$a < b$	$\rightarrow$	*a* yields precedence to *b*
$a = b$	$\rightarrow$	*a* has same precedence *b*
$a > b$	$\rightarrow$	*a* takes precedence over

For the example *id* + *id* * *id*.

Operator Precedence Relations

	id	**+**	*****	**$**
id	–	· >	· >	· >
+	< ·	· >	< ·	· >
*	< ·	· >	· >	· >
$	< ·	< ·	< ·	–

Then the string with the precedence relations inserted is:

$ < · id · > + < · id · > * < · id · > $

For example < · is inserted between the leftmost $ and id since < · is the entry in row $ and until (3) column id.

The handle can be found by the following process:

1) Scan the string from the left end until the first · > is encountered. In the above, case this occurs between the first id and '+'.
2) Then scan backwards until < · is encountered.
3) Symbol between < · and · > is the handle.

   Example: id as in the above case.

If we are dealing with the preceding grammar we then reduce id to E. At this point we have the right sentential form E + id * id. After returning the two remaining id's to E by the same steps we obtain the right sentential form E + E * E.

The precedence relations indicate that in the right sentential form E + E * E, the handle is E * E. Since '*' has got higher precedence than '+'.

Disadvantage of this method of finding the handle:

- The entire right sentential form must be scanned at each step to find the handle.

Solution:

- Using stack to store input symbols.
- Using precedence relation to guide the actions of a shift reduce parser.

  If precedence relation < · and = · holds between the topmost terminal symbol on the stack and the next input symbol, the parser shifts.
- If the relation · > holds the right end of the handle is found and a reduction is called for

- If no precedence relations holds between a pair of terminals then a syntactic error has been deleted and an error recovery routine must be invoked.

Operator Precedence Parsing Algorithm

Input: An input string ω and a table of precedence.

Output: If 'ω' is well formed, a skeletal parse tree with a non-terminal E labeling all interior nodes, otherwise an error indication.

Method: Initially the stack contains $ and the input buffer the string ω$. To parse we execute the following program.

```
1)  Set input to point to the first symbol of ω$;
2)  repeat
3)  if $ is start symbol, is an top of the stack and input points to $ then
4)  return
    else begin
5)  Let 'x' be the topmost terminal symbol on the stack and
          'y'n be the symbol pointed to by input
6)  If x <.y or x $\overset{\cdot}{=}$y then begin
7)  Push b onto the stack;
8)  Advance input to the next input symbol;
    end;
9)  else if x · > y then
10) repeat
11) POP the stack
12) until the top stack terminal is related by < · to the terminal most recently popped and
    reduce.
13) else error ()
14) end if
```

Stack	**Relation**	**Input**	**Handle**	**Action**
$	<·			Shift

$id	·>	id + id * id$	id	Reduce POP
$	<·	+ id * id$		Shift
$+	<·	id * id$		Shift
$+id	·>	* id $	id	Reduce POP
$+	<·	* id $		Shift
$+*	<·	id $		Shift
$+*id	·>	$	id	Reduce POP
$+*	·>	$	E * E	Reduce POP (since '*' is present on the right side of the production E $\rightarrow$ E * E)
$+	·>	$	E + E	Reduce POP (since '+' is present on the right side of the production E $\rightarrow$ E + E) accept.
$	–	$		accept

2.9.1 BOTTOM UP PARSING

A general style of bottom up syntax analysis is known as shift reduce parsing. Shift reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. This process reduces a string 'ω' to the start symbol of a grammar.

At each reduction step, a particular substring matching the right side of a production is replaced by the symbol on the left of the production.

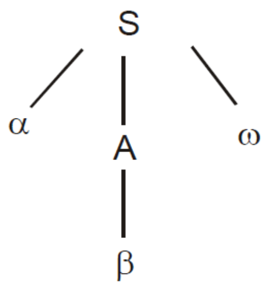
For example, consider the grammar:

$$S \rightarrow aABe \ A \rightarrow Abc|b \ B \rightarrow d$$

The sentence abbcde can be reduced to 'S' by the following steps in the example, abbcde is a right sentential form whose handle is A $\rightarrow$ *b* at position 2.

Like wise *a*A*bcde* is the right sentential form whose handle is A $\rightarrow$ A*bc*.

The handle A → β is represented in the following parse tree for αβω.



In this case, reducing β to A in αβω can be thought of as "pruning the handle" or removing the children of A from the parse tree.

Consider the following grammar:

S → *a* A B *e*

A → A *b c* | *b*

B → *d*

$a\underline{b}bcde$

$aA\underline{bc}de$

*a*A*de*

*a*AB*e*

S

Thus by a sequence of four reductions, we are able to reduce abbcde to 'S'. Those reductions in fact trace out the following right most derivation in reverse.

$$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$$

Handle

A handle of a right sentenial form γ is a production A → B and a position of 'γ' where the string B may be found and replaced by A to produce the previous right sentenial form in a right most derivation of γ.

If $S \underset{rm}{\overset{*}{\Rightarrow}} a\,A\omega \underset{rm}{\Rightarrow} \alpha\beta\omega$ then A′ → β is a handle of αβω and the rightmost derivation:

$$E \underset{rm}{\Rightarrow} E + E \qquad\qquad E \Rightarrow id_1 + id_2 * id_3$$

$$\underset{rm}{\Rightarrow} E + E * E \qquad\qquad \Rightarrow E + id_2 * id_3$$

$\underset{rm}{\Rightarrow} E + E * id_3$ $\Rightarrow E + E * id_3$

$\underset{rm}{\Rightarrow} E + id_2 * id_3$ $\Rightarrow E + E * E$

$\underset{rm}{\Rightarrow} id_1 + id_2 * id_3$ $\Rightarrow E * E$

$\Rightarrow E$

Here, id1 is the handle of the right sentential forms id1 + id2 * id3 because *id* is the right side or production E → id and replacing id1 by E produces the previous right sentential form:

E + *id* **id* 3

Since the grammar is ambiguous, there is another rightmost derivation of the same string.

$$E \underset{rm}{\Rightarrow} E * E$$

$$\underset{rm}{\Rightarrow} E * id_3$$

$$\underset{rm}{\Rightarrow} E + E * id_3$$

$$\underset{rm}{\Rightarrow} E + id_2 * id_3$$

$$\underset{rm}{\Rightarrow} id_1 + id_2 * id_3$$

Handle Pruning

The rightmost derivation in reverse can be obtained by 'handle puring'. The process is repeated until the right sentential form consists of only the start symbol. The reverse of the sequence of productions used in the reductions is a rightmost derivation of the input string.

Example: The sequence of steps of the reduction of the input string *i* + *i* * *i* is shown below table.

NOTE: This is just the reverse of the sequence in the rightmost derivation sequence.

The grammar is:	**The right most derivation is:**
E → E + E	E → E + E
E → E * E	E → E + E * E
E → (E)	E → E + E * *i*
E → *i*	E → E + *i* * *i*
	E → *i* + 1 * *i*.

Right Sentential Form	*Handle*	*Reducing Production*
i + *i* * *i*	*i*	$E \rightarrow i$
E + *i* * *i*	*i*	$E \rightarrow i$
E + E * *i*	*i*	$E \rightarrow i$
E + E * E	E * E	$E \rightarrow E * E$
E + E	E + E	$E \rightarrow E + E$
E		

Problems that must be solved if we are to parse by handle pruning:

- Locate the substring to be reduced in a right sentenial form.
- Determine what production to choose in case there is more than one production with that substring on the right hand side.

2.9.2 OPERATOR PRESCEDENCE RELATION

Operator Precedence Relation must be created, such that the operator precedence parsing algorithm will work correctly when guided by them.

The following rules are designed to select the proper handles to reflected a given set of associativity and precedence rules for binary operators.

1) If operator θ_1 has higher precedence than operator θ_2 make $\theta_1 \cdot > \theta_2$ and $\theta_2 \cdot < \theta_1$.

   For example if * has higher precedence than + make * · > + and + < · *. These relations ensures that in an expression of the form "E + E * E".

   "E * E" is the handle that will be reduced first.

2) If θ_1 and θ_2 are operator of equal precedence make $\theta_1 \cdot > \theta_2$ and $\theta_1 \cdot > \theta_2$ if the operator are left associative or make $\theta_1 <\cdot\ \theta_2$ and $\theta_2 <\cdot\ \theta_1$ if they are right associative.

   Example: If +, – are left associative then make + · > +, + · > –, – · > –, and – · > +.

   This will ensure "E – E" selected in E – E + E.

   If ↑ is right associative make ↑ < · ↑ .

   This will ensure the last E ↑ E in E ↑ E ↑ E to be selected.

3) Make θ < · id, id · > θ

   For all operators θ

$\theta < \cdot ($

$(< \cdot \theta$

$) \cdot > \theta$

$\theta \cdot >)$

$\theta \cdot > \$$

$\$ < \cdot \theta$

Also let,

$(=)$

$(< \cdot ($

$(< \cdot \text{id}$

$\$ < \cdot ($

$\text{id} \cdot > \$$

$\text{id} \cdot >)$

$\$ < \cdot \text{id}$

$) \cdot > \$$

$) \cdot >)$

2.10 LR PARSER

- The bottom up syntax analysis technique that can be used to parse a large class of context free grammar is called LR (K) parsing.

  'L' stands for left to right scanning of the input.

  'R' for constructing a right most derivation in reverse.

  'K' is the number of input symbols of look ahead.

2.10.1 FEATURES OF LR PARSER

- It can be constructed to recognize all programming language constructs for which CFG's can be written.
- It is the most general non-backtracking shift reduce parsing method known.
- Grammars that can be parsed using LR method can be parsed with predictive parses.

The schematic form of LR parser is shown in the Figure.

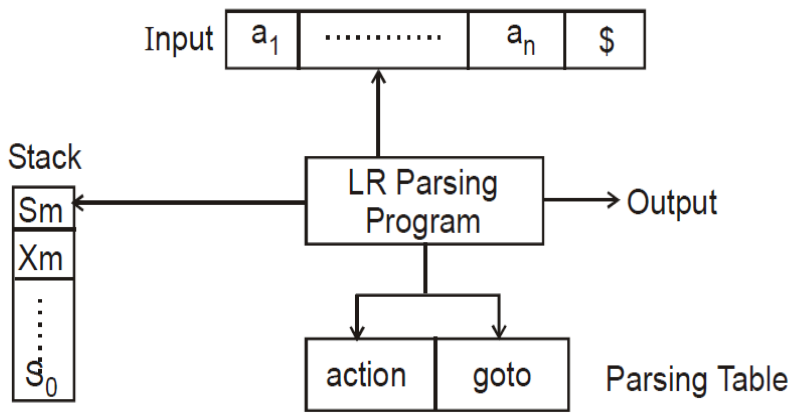


Figure 2.16

The LR parser consists of:

1) Input buffer
2) Output
3) Stack
4) Driver program/parsing program
5) Parsing table that has two parts (action and goto).

There are three ways to construct the LR passing tables as listed below:

1) SLR (Simple LR) (LR(0))
2) CLR (Canonical LR) (LR(1)) and
3) LALR (Look ahead LR) (LR (1))

The parsing program is common for all the 3 cases, but the parsing lable changes from one parser to another.

SLR Parser

In order to construct the SLR parsing table the following two components are needed.

1) Construction of sets of LR(0) items collection using, CLOSURE function and GOTO function.
2) Construction of parsing table using LR(0) items construction of LR(0) items.

Construction of LR(0) items

LR(0) item of a grammar G is a production of G with a dot at some position on the right side of the production.

Example: if $E \rightarrow X$, is a production, the LR(0) items are $E \rightarrow \cdot X$ and $E \rightarrow X \cdot$

The collection of sets of LR(0) items must be constructed the in order to construct SLR parsing table.

The collection of sets of LR(0) items can be constructed with the help of functions called closure and goto. The collection of sets of LR(0) items is called canonical collection of LR(0) items.

The Closure Function

Let us say 'I' is a set of items for a grammar G, then closure of I (CLOSURE (I)) can be computed by using the following steps.

1) Initially every item in I is added to closure (I).
2) Consider the following:

   $A \rightarrow X \cdot BY$ an item in I

   $B \rightarrow Z$ a production.

Then add the item $B \rightarrow Z$ also to the closure (I). If it is not already there, this rule has to be applied till no new items can be added to closure (I). (That is if there is a '.' before a non-terminal then include the production rules of that non-terminal also with a dot in the right side of the production).

GOTO FUNCTION

This is computed for a set R on a grammar symbol X.

If $A \rightarrow \cdot X\ BY$ is in I (item)

goto (I, X) will be $\Rightarrow$ $A \rightarrow X \cdot BY$,

Even before the closure function is computed, the augmented grammar has to be found, as a first step in the construction of the canonical collection of LR(0) items for the given grammar G.

Augmented Grammar

Consider the grammar G, in which ‘S’ is the start symbol. The augmented grammar of G is G′ which a new start symbol S′ and having a production rules of the given grammar G.

Example 1: Construct the canonical collection of LR(0) items for the given grammar G.

$E \to E + T$

$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to id$

Augmented grammar ‘G”

$E' \to E$

$E \to E + T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to id$

The next step is to find closure ($E' \to . E$)

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

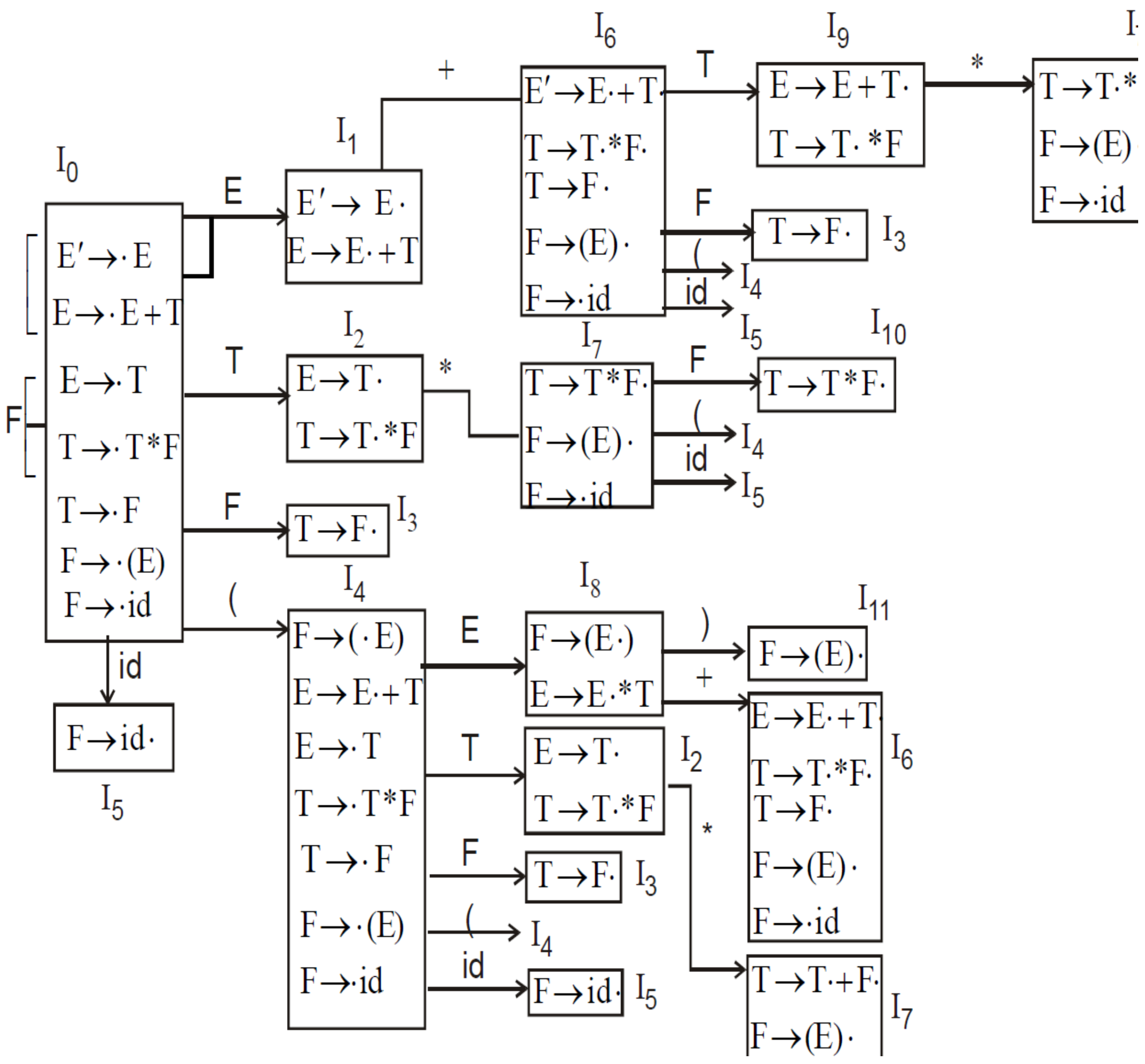


Figure 2.17

//as there is a '.' before 'T' include the production rules of T with a dot is the right hand side of " → "

//as there is a '.' before 'F' include the production rules of 'F' with a dot in the right hand side of " → "

NOTE:

1) Stop when there are no more new states on which the goto function might be applied.
2) When . is followed by NT, include the production of that NT.

2.11 SLR PARSING TABLE CONSTRUCTION

This is also a 2 Dimensional array in which the rows are states end columns are terminals and non-terminals.

The label has 2 parts as:

1) Action entries (which consist of terminals).
2) Goto entries (which consist of non-terminals).

The action may be any one of the follow:

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

The goto entries will have state numbers like 1, 2, 3.

State: Consider a set I_j in the collection of LR(0) items. Here '*j*' is the state which is used in parsing the input string. **Example:** If I_i is an item '*i*' is the state.

Goto Entries.The goto entries indicate the transition of a state '*i*' to '*j*' on a particular non-terminal.

Steps for the construction of the parsing table

1) Fill up the 'goto' part which the next state for each non-terminal.

   Example: I_0 on 'E' transition is I1 so enter goto (0, E)=1.

   – Repeat for all states.

2) M[1, $] = accept (by default).

3) To make shift entries fill up the action part with Si where 'I' is the next state for each terminal.

   Example: I_i on '+' transition I_6 so, enter action (1+) = S_6.

   – Repeat for all states.

4) States with (.) as the right most symbol in the corresponding item will have reduce entries. If A $\rightarrow$ XBY. is present in the item, and 'Y' is a terminal take follow (A), on the other hand if '*y*' is a non-terminal take follow (Y) fill in the cell corresponding to the respective state and elements in follow (A) or follow (Y) with r_j where '*r*'refers to reduce and '*j*' refers to the rule number.

   Example: I_2 has E$\rightarrow$T as E$\rightarrow$T' is the 2nd production rule in the grammar which follows:

   (1) $E \rightarrow E + T$ (2) $E \rightarrow T$ (3) $T \rightarrow T \times F$

(4) $T \rightarrow F$ (6) F (5) $F \rightarrow (E)$

$\rightarrow$ id

Note: FOLLOW(E) ={+,$.)}

Enter r_2 under the elements of follow(E), where γ is reduction and 2 is the 2nd production in the unaugmented list.

All the undefined entries are errors.

Parsing Table

Table

State	**Action**						**Goto**		
	Id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**E**
0	S_5		S_5				1	2	3
1		S_5				Accept			
2		r_2	S_5		r_2	r_2			
3		r_2	r_2		r_2	r_2			
4	S_5		S_5				8	2	3
5		r_2	r_2		r_2	r_2			
6	S_5			S_5				9	3
7	S_5			S_5					10
8					S_5				
9		r_2	S_5		r_2	r_2			
10		r_2	r_2		r_2	r_2			
11		r_2	r_2		r_2	r_2			

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

After eliminating left recursion.

$E \rightarrow TE''$

$E' \rightarrow + TE'|\varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT'|\varepsilon$

$F \rightarrow (E)/id$

FIRST (E) = {(, id}

FIRST (T) = {(, id}

FIRST (F) = {(, id}

FIRST (E′) = {+, $}

FIRST (T′) = {*, $}

FOLLOW (E)

Consider $F \rightarrow (E)$

FOLLOW (E) = FIRST ( )) = {)}

$\because$ E is the start symbol

FOLLOW (E) = {2, 3}

Consider $\boxed{E \rightarrow TE'}$

FOLLOW (T) = FIRST (E′) + FOLLOW (E)

$\therefore$ {+,$,)}

$\boxed{T \rightarrow FT'}$

FOLLOW (F) = FIRST (T′) + FOLLOW (T)

= {*, $, +,)}.

PARSING ALGORITHM

For parsing an input string. The possible parsing actions are as follows:

1. Shift
2. Reduce by a production
3. Accept and halt
4. Error

The input string is in input buffer followed by the right end marker $. The stack keeps the states of the Parsing Table.

Steps involved in SLR Parsing

Initially action [stack top symbol, input buffer 1st symbol] is referred in the parsing table.

1) If action [S_x , a_y] = S_j , the parser has to make a shift of the current input symbol from

the buffer to the stack, and then push '*j*' also onto stack.

2) If action [S_x , a_y] = r_j ; the parser has to make a reduction by the rule number *j*.

   If the reduction rule is of the form $\alpha \rightarrow \beta$, in this case, the top $\boxed{\therefore |\beta| * 2}$ elements are popped from the stack the reduction is applied for the popped elements and the resulting, element is pushed onto the stack, then, the table is referred for goto (stack [top-1], stack [top]). The referred symbol there after pushed onto the stack.

   Note: |β| refers to the number of elements in the right side of the production.

3) If Action [Sx, *ay*] = accept, then announce that the parsing is completed successfully and then halt.

4) If Action [Sx , *ay*] = error, then the parser encounters error and calls error recovery routine or generates error message.

STACK	INPUT	ACTION	
0	id + id$	action (0, id) = S_5	[Note S denotes Shift Operation
0id5	+ id$	action (5, +) = r_6	Hence Shift id 5]. [F→*id*]
0F3 →[Refer goto(0,T) in the parsing table]	+ id$	action (3, +) = r_4	$\|id\| = 1$ pop 2 elements
0T2	+ id$	action (2, +) = r_4	push F onto the stack.
0E1	+ id$	action (1, +) = S_6	Then Refer 0, F on to the
0E1 + 6	id$	action (b, d) = S_5	Parsing Table which is 3
0E + 6 id5	$	action (5, $) = r_6	Push 3]
0E1 + 6F3→[Refer goto(6,F)]	$	action (3, $) = r_4	
0E1 + 6T9	$	action (9, $) = r_1	∵ E → E + T
0E1	$	action (1, $) = accept	

Disadvantages of SLR Parser

- Too much work to construct the parser.
- Some time there may be both a shift and reduce entry in action [X, Y]. This conflict arises from the fact that the SLR parser construction method is not powerful enough to remember.

2.11.1 OPERATOR PRECEDENCE RELATIONS FOR THE GRAMMAR

$E \rightarrow E + E|E - E|E * E|E/E|E \uparrow E|(E)| - E|id$

Using the stated rules is shown below:

	+	-	*	↑	*	Id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Note

If ¬ [logical negation) is a unary prefix operator, we make

$\theta <\cdot\ \neg$ for any operator θ whether unary or binary

$\neg \cdot> \theta$ if ¬ has higher precedence than θ and

$\neg <\cdot\ \theta$ if ¬ has lower precedence than θ.

Sometimes we may have an operator like '–' that is both unary and binary.

In such situations the lexical analyzer distinguish between unary and binary minus.

For example a minus sign is unary if the previous token was an operator, a left parenthesis, a comma, or an assignment symbol (in Fortran).

2.11.2 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 '*x*' and '*y*'.

(1) $f(x) < g(y)$ Whenever $x < \cdot\, y$

(2) $f(x) = g(y)$ Whenever $x =\cdot\; y$

(3) $f(x) > g(y)$ Whenever $x \cdot > y$.

Thus the precedence relation between 'y' can be determined by a numerical comparison between $f(x)$ and $g(y)$.

Note: That the error entries in this precedence matrix are not clear, since one of (1), (2) or (3) holds no matter what $f(a)$ and $g(b)$ are, but this defect is not serious as errors can still be caught when a reduction a called for and no handle can be found.

Algorithm: Constructing precedence functions.

Input: An operator precedence matrix.

Output: Precedence functions representing the input matrix or an indication that none exist.

Method:

(1) Create symbols *fa* and *ga* for each '*a*' that is a terminal or $.

(2) Partition the created symbols into as many groups as possible in such a way that

* if $a \doteq b$ and $c \doteq b$ then *fa*, *fc* are in the same group as *gb*.

* If $a \doteq b$ and $c \doteq b$ and $c \doteq d$ then *fa* and *gd* are in the same group.

(3) Create a directed graph whose nodes are the groups found in (2)

Example: If $a < \cdot\, b$ place an edge from *gb* to *fa*.

If $a \cdot > b$ place an edge from *fa* to *gb*.

(4) If the graph constructed in (3) has cycle then no precedence functions exist.

If there are no cycles let

$f(a)$ be the length of the longest path beginning at the group of *fa*.

Let $g(a)$ be the length of the longest path from the group of *ga*.

Example: Operator precedence relations if $fa > gb$ $a \to b$

(g) $fa < gb$ $b \to a$

	id	+	*	$
id	–	· >	· >	· >
(f)+	< ·	· >	< ·	· >
*	< ·	· >	· >	· >
$	< ·	< ·	< ·	–

In this case there is no = relationship so each symbol is in a group by itself. The corresponding graph constructed using the preceding algorithm is:

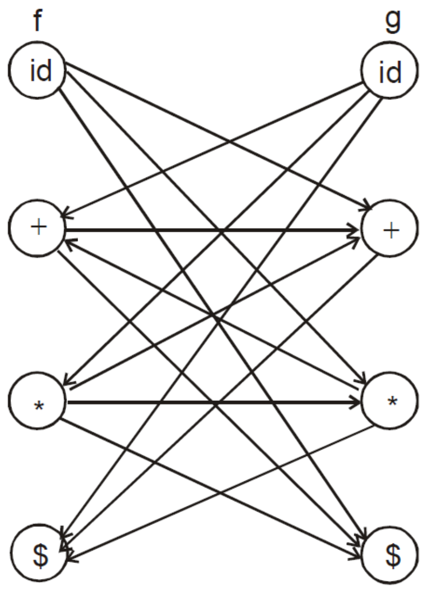


Figure 2.15

Note:

1) When $fa \cdot > gb$ draw an edge from a to b

2) When $fa < \cdot\ gb$ draw an edge from b to a

3) When $fa = gb$ draw edges from a to band 'b' to 'a'.

Graph representing Precedence Functions

NOTE:

There are no cycles so, precedence function exists.

As f\$ and g\$ have no out edges $f(\$) = g(\$) = 0$.

The longest path from $g(+) = 1$(length) so $g(+) = 1$ (since there is only one edge from $g + t\ f$ $).

There is a path from $gid \to f* \to g* \to f+ \to g+ \to f\$$ so

$g(id) = 5$. The resulting precedence functions are:

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

Error Recovery in Operator Precedence Parsing:

There are two points in the parsing process at which an operator precedence parser can discover syntactic errors:

1) If no precedence relation holds between the terminal on top of the stack and the current input.
2) If a handle has been found but there is no production with this handle as a right side.

Handling Errors during Reductions:

Error detection and recovery routine is divided into several pieces. One piece handles errors of Type (2). For example suppose *abc* is popped and there is no production right side consisting of *a, b* and *c*. Then we might consider if deletion of one *a, b* and *c* yields a legal right side.

- We might also consider changing or inserting a terminal.
- We may also find that there is a right side with the proper sequence of terminals but the wrong pattern of non-terminals. For example, if *abc* is popped off the stack and abc is not a right side but *aEbc* is we might issue a diagnostic missing E on line (line containing *b*).

  For the grammar E → E + E | E – E | E * E | E/E | E ↑ E | (E) | – E | id.

Syntax checker does the following:

(1) If +, –, *, /, or ↑ is reduced.

→ it checks that non-terminal appear on both sides. If not it issues the diagnostic missing operand.

(2) If *id* is reduced it checks that there is no non-terminal to the right or left.

→ If there is it can warn missing operator.

(3) If ( ) is reduce, it checks that there is a non-terminal between the parenthesis.

→ If not it can say

no expression between parenthesis.

Handling Shift/Reduce Errors

When consulting the precedence matrix to decide whether to shift or reduce, we may find that no relation holds between the top stack symbol and the 1st input. To recover we must modify the stack, input or both we may change symbols, insert symbols onto the input or stack or delete symbols from the input or stack.

Disadvantage of Operator Precedence Parsing

- It is hard to handle tokens like the minus sign which has two different precedence. (depending on whether it is unary or binary).
- Only a small class of grammars can be parsed using operator precedence techniques.

Advantages:

- Simplicity.

2.12 LALR PARSER

The look ahead LR parser is another parser in the LR parser category. This parser also constructs the parsing table from LR(1) items. There is a slight modification in the construction of the parsing table, for LALR parsers, and the parsing algorithm is very much same as that of the other LR parsers.

LALR are same as CLR parser. In CLR parser, if two states differ only in lookahead, then we combine those two states in LALR parser. After minimization, if the parsing table has no conflict then that grammar is LALR.

Steps for constructing LALR Parsing Table:

1) C = {I_0 , I_1 , ..., I_n} be the collection of LR(1) items.
2) Find the sets having core elements present in collection of LR(1) items and replace them by their unions. (i.e) If I_i and I_j have the same core items. They can be united as I_{ij} .
3) All the remaining steps are similar to the construction of the CLR parsing table.
4) Lets consider the same problem, discussed in CLR parser

I_{36}	:	$C \rightarrow C.C$, cld|\$ $C \rightarrow .cC$, cld|\$ $C \rightarrow .d$, cld|\$	Since I_3 and I_6 have the same core elements they are united as I_{36} (Table of CLR in previous page).
I_{47}	:	$C \rightarrow d.$, c|d|\$	Since I_4 and have seen core elements they are united as I_{47}.
I89	:	cC., cd, \$	Since Is and Io have the same care elements thus are united as I_{89}.

Table 2.2: LALR Tables

States	Action			Goto	
	c	**d**	**\$**	**s**	**e**
0	S_{36}	S_{47}		1	2
1			accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5	X		r_1		
89	S_6	r_2	r_2		

Grammar is $S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$.

Stack	I/O	Action
0	cdd \$	S_{36}
$0C_{36}$	dd \$	S_{47}
$0C_{36}d_{47}$	d\$	r_3 || reduce by production rule 3 “$C \rightarrow d$”
$0C_{36}$ C_{89}	d\$	r_2 || reduce by production rule 2 “$C \rightarrow cC$”

$0C_2$	d$	S_{47}
0C2*d*47	$	r_3 || reduce by production rule 3 "C→d"
0C2Cs	$	r_1 || reduce by production rule 1 "S→CC"
0S1	$	accept

NOTE:

- Wherever S_3 or S_i occured in Parsing Table of CLR action entry 'S_{36}' will now appear.
- Similarly wherever S_4 or S_7 appeared in CLR action entry 'S_{47}' will now take over.
- Reduce entries of state '8' and '9' are merged together.
- Reduce entries of state '4' and '7' are merged together in one row.

2.12.1 CLR PARSER

It is also an LR parser, CLR is canonical LR parser. Many of the concepts in CLR parser are similar to the SLR Parser but there are some differences in the construction of parsing table.

- The construction of parsing table is done with LR(1) items.
- The steps involved for construction of parsing table (T) from the LR(1) items are also different from the SLR table construction.

LR(1) Items: where 1 is the item of the second component.

The general term of LR(1) item is $A \rightarrow X . Y, a$; where 'a' is called look ahead. It may be a terminal or the right end marker $.

Example: $S' \rightarrow S \cdot \$$, where $ is the look ahead.

The collection of LR(1) items will lead to the cosntruction of LR(1) items will lead to the construction of CLR parsing table.

The Closure Function:

Let us say 'L' is a set of LR(1) items for the given grammar, then closure of I represented as (Closure (I)) can be computed using the following steps:

1) Initially every item in R is added to closure (R).
2) Consider, $A \rightarrow X.BY, a$

$$B \rightarrow Z$$

are the two production and X, Y, Z are grammar symbols. Then add B → . Z, First (Y) as the new LR(1) if its not already there.

This rule has to be applied till no new items can be added to the closure (R). Thus the closure (A → X . BY, *a*) will have,

$$A \rightarrow X\ .\ BY,\ a$$

$$B \rightarrow .\ Z,\ \text{FIRST}\ (Ya).$$

***Note*:**

1) If Y is a terminal or a non-terminal the look ahead will be, FIRST (Y).
2) If Y is not available then the look ahead will be FIRST (a).

GOTO FUNCTION:

This is computed for a set 'I' on a grammar, what is the set 'I' reaches on X can be computed by the function:

goto (I, X)

Consider an LR(1) item is 'I' which follows:

$$A \rightarrow \cdot XBY,\ a$$

goto (I, x) will be,

$$\boxed{A \rightarrow X \cdot BY,\ a}$$

including the closure of B′ and its look ahead where X, Y are grammar symbols.

Example: Construct the LR(1) items for the following grammar.

$$S \rightarrow CC$$

$$C \rightarrow cC$$

$$C \rightarrow d$$

Solution:

Step 1

The augmented grammar G′ is:

$$S' \rightarrow \cdot S$$

$$S \rightarrow \cdot CC$$

$$C \to \cdot cC$$

$$C \to \cdot d$$

Step 2

Add the second component.

The second component is added to avoid S/R conflict.

S/R Conflict	**Explanation**
I. $S' \to \cdot S, \$$	[Add \$ as second component to the I production in G']
II. $S \to \cdot CC$ $S \to \cdot CC\ , \$$	[To find the second component make use of the I production for comparison with the standard format $A \to \alpha \cdot B\beta, a$ $S \to \cdot S, \$$ β is $\in$, a is \$. Second component b is First ($\in$ \$) First (\$) = \$
III. Consider the production $C \to .cC$ $C \to .cC , c \mid d$	$A \to \alpha \cdot B\beta, a$ Second component is $S \to \cdot CC, \$$ First (C) = c|d.
IV. $C \to .d$ since LHR is C. Copy c | d to the second component $C \to .d, c \mid d$	Second Component is First (β, a) First (c) = c | d \$ is not included because First (c) has terminals c, d since c does not derive empty string. First ((, \$) = First (c)

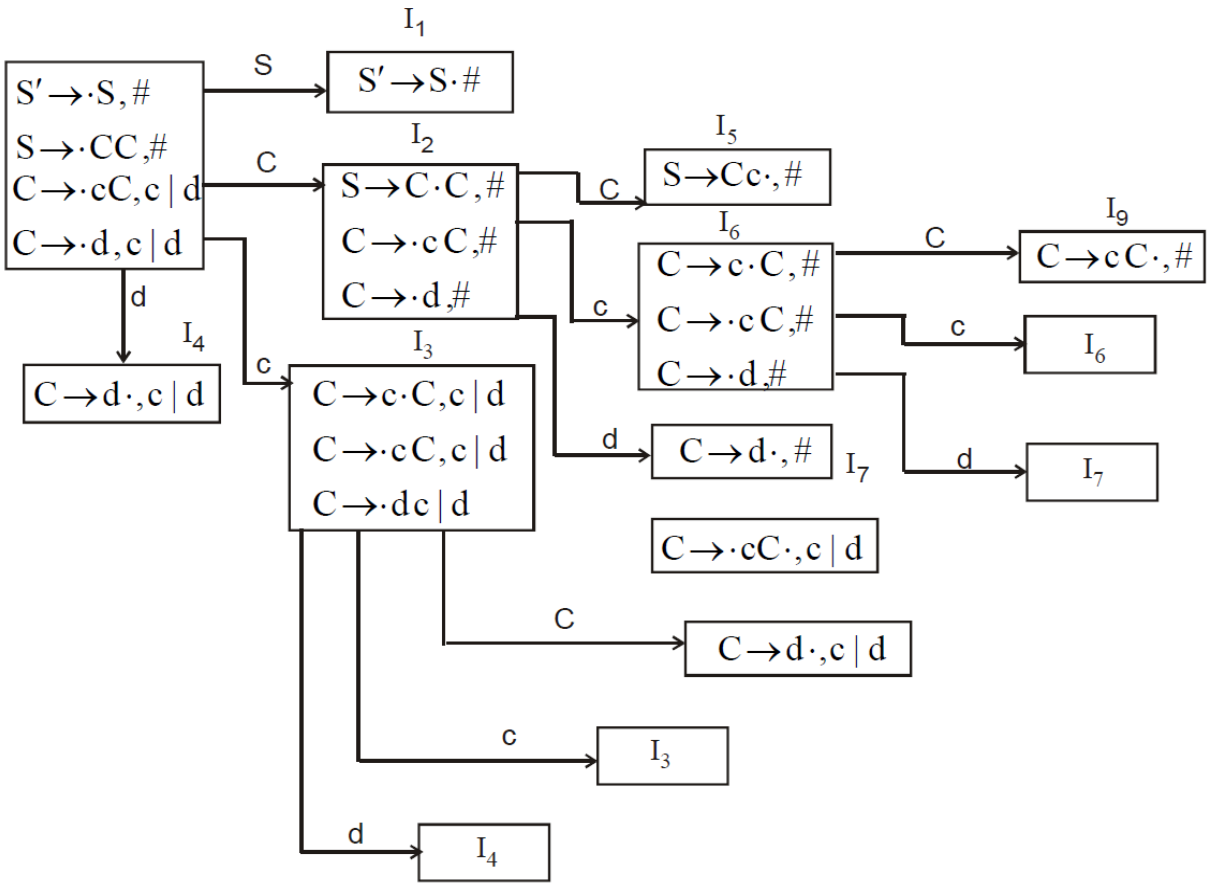


Figure 2.18

Note: Consider the state I_2, with the second component C → · cC, $.

In this original production the II component is c | d.

The I Rule's second component is to be taken for the remaining production in the state.

CLR Parsing Table Construction:

This is also a two dimensional array, in which the rows are states and columns are terminals and non-terminals. The table has two entries namely.

1) Action Entries
2) Goto Entries.

Steps for the Construction of CLR Parsing Table:

1) Let C = {I_0 , I_1 , I_2 , ..., I_n} be a collection of sets or LR(1) items
2) Consider I_j as a set in C.
   a. If goto (I_j , a) = J_k then set action [*j*, *a*] to S_k and here 'a' is always a terminal.

b. If A $\rightarrow$ X . , a (X is a grammar symbol) is in Ij, then set action [*j*, *a*] to reduce by A $\rightarrow$ X here '*a*' is look ahead and 'A' should not be S′.

c. Set action [1, $ as accept]

[$\because$ S′ $\rightarrow$ S . is in I1].

3) If goto [I_j , A] = I_k then set goto [*j*, A] = *k*.

4) All the undefined entries are errors. In the case of CLR parsing technique the reduce entries are made for look ahead terminals.

S $\rightarrow$ CC

C $\rightarrow$ *c*C

C $\rightarrow$ *d*

CLR Parsing Table

States	**Action**			**Goto**	
	c	**d**	**$**	**s**	**e**
0	S_3	S_4	$	1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_3			8
4	r_3	r_3			
5			r_1		
6	S_6	S_6			9
7			r_3		
8	r_2	r_2			
9			r_2		

Stack	**I/O**	**Action**
0	cdd $	S_3
$0C_3$	cdd $	S_4
$0C_3d_4(4)$	d$	r_3 || reduce by production rule 3 C$\rightarrow$d

$0C_3$	d$	r_2 c →cC [pop 4 elements push C, Goto (0, E) = 2; push 2]
$0C_2$	d$	S_7 refer goto (3, C) from the table
0C2*d*7	$	r_3 [c → d pop out 2, elements push C, Goto (2, C) = 5]
0C2C5	$	r_1
0S1	$	accept

2.13 ERROR RECOVERY STRATEGIES

There are many different general strategies that a parser can employ to recover from a syntactic error, they are:

- Panic Mode Error Recovery
- Phrase Level Error Recovery
- Error Productions
- Global Correction.

2.13.1 PANIC MODE RECOVERY

- It is the simplest method to implement and can be used by most parsing methods.
- On discovering an error the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synthesizing tokens are usually delimiters such as semicolon or end.
- Panic mode correction often skips a considerable amount of input without checking it for additional errors.

Advantage:

- Simplicity
- Guaranteed not to go into an infinite loop.

Very useful when multiple errors in the same statement are rare.

2.13.2 PHRASE LEVEL RECOVERY

- On discovering an error a parser may perform local correction on the remaining input that is it may replace a prefix of the remaining input by some strings that allows the parser to continue.

  Example: By replacing ‘,’ by

  - Inserting a missing semicolon,

- o Deleting an extraneous ';'.

This type of replacement has been used in several error repairing compilers.

Disadvantage:

Difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error Productions Recovery

If we have good idea of the common errors that might be encountered we can augment the grammar for the language at hand with productions that generate the erroneous constructs. We then use the grammar with the error productions to construct a parser.

- If an error production is used by the parser, we can generate the appropriate error diagnosis and recovery mechanisms.

Global Correction Recovery

There are algorithms for choosing a minimal sequences of changes to obtain a globally least cost correction.

Given a incorrect input string *x* and grammar G, these 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.

Disadvantage:

- Too costly.

Note: The closest correct program may not be what the programmer had in mind, after these error recovery strategies are applied.

2.14 INTRODUCTION TO YACC

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:

YACC input file is divided in three parts.

```
/* definitions */
```

```
 ....
%%
/* rules */
....
%%
/* auxiliary routines */
....
```

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:
- %token NUMBER

```
%token ID
```

- Yacc automatically assigns numbers for tokens, but it can be overridden by

```
%token NUMBER 621
```

- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within **%{**and **%}** in the first column.

Input File: Rule Part:

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:

```
#include "lex.yy.c"
```

- YACC input file generally finishes with:

```
.y
```

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the **main()** definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function **int yyparse()**
- If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v** option, Yacc produces as output a file**y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.