

Translation Design

Lexical analysis (scanner)	تحليل المفردات
Parsing (syntax analysis)	الاعراب
Semantic analysis	تحليل دلالي
Translation to intermediate mode	التحويل الى الصيغة الوسيطة
Code generation	الحصول على الايعازات
Code optimization (Loop optimizations)	تحسين البرنامج
Algorithm for instruction selection	خوارزميات ربط الجمل

المصادر:

- Kenneth C.Louden 2005-2006 “ Compiler Construction Principles & Practice “
- R. Wilhelm. “ Compiler Design “ Addison _ Wesley , 1995
- V. Aho. R, Sethi , J.D. Ulman , “ Compiler Principles , Techniques & Tools “ , Addison Wesley , 1986
- J.P. Tremblay, P.G Sorensen, “The Theory & Practical of computer writing”, Megraw Hill, 1985.

Compilers:

A compiler is a program that reads a program written in one language, the source language, and translates it into an equivalent program in another language, the target language (fig 1.1)

as an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

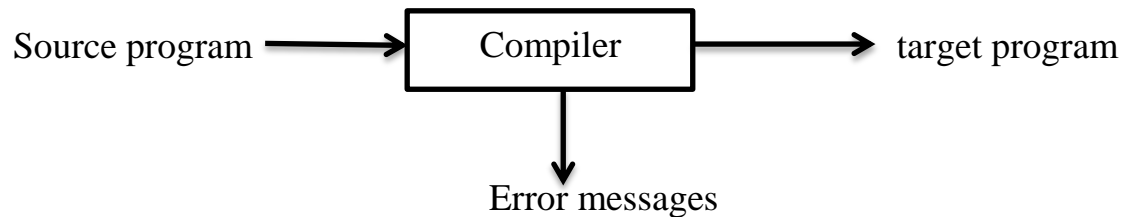


Fig.1: a compiler black box

Definitions:

Translator: is a program that converts a source program into an object program.

Assembler: is a translator that converts source program written in assembly language to machine language

Compilers: is a translator that transforms HILL such as Fortran or Pascal or Cobol, into an assembly language. The time at which the conversation of a source program to an object program occurs is called compiler time. The object program is excited at run time.

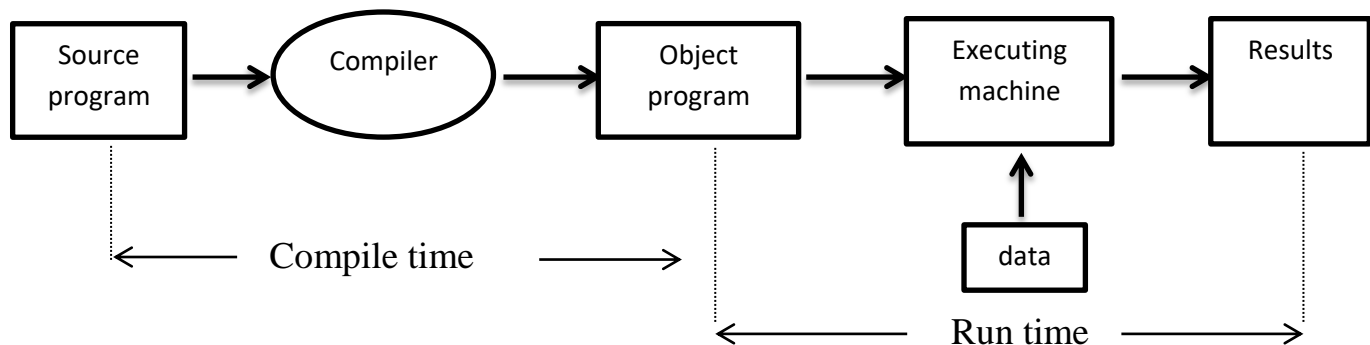


Fig _2. Compiler

Interpreters: is a translator that process source program and data at the same time that is interpretation of the source from accurate at runtime and no object program is generated.

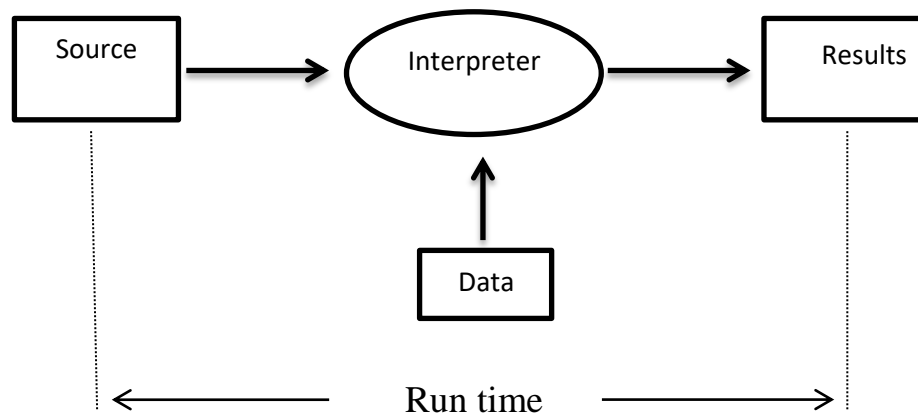
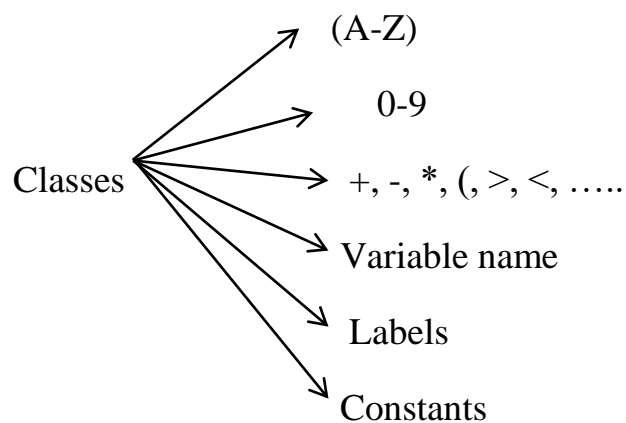
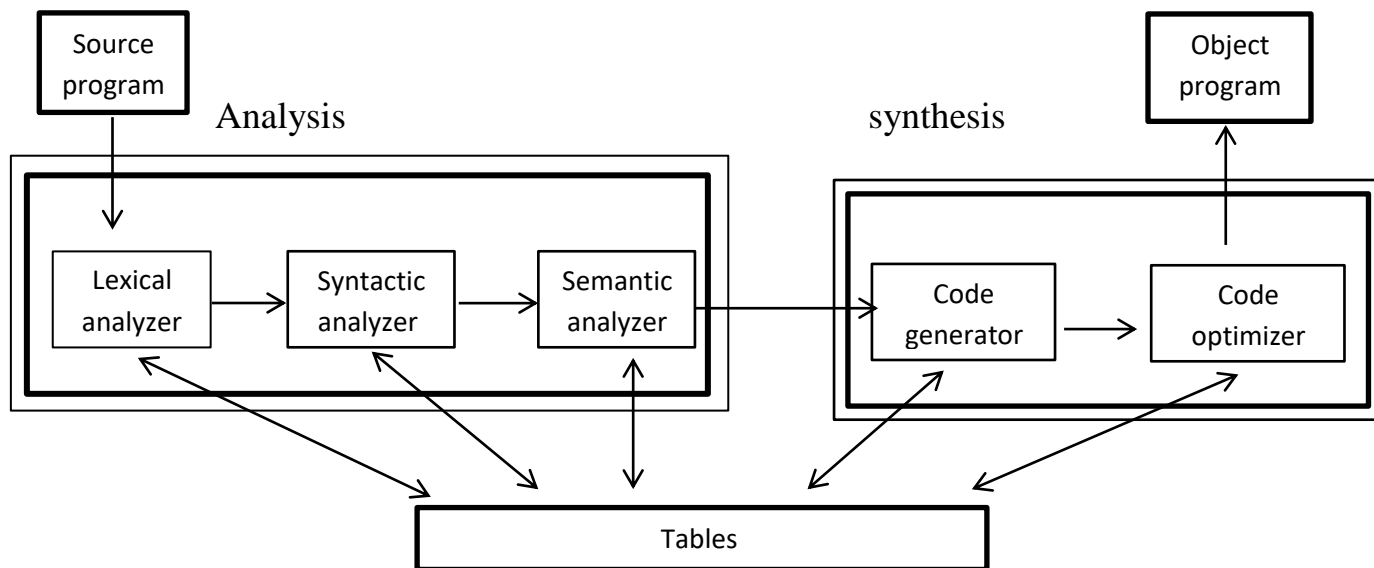


Fig.3_ Interpreter

Source program: is a string of symbols which is a letter, a digit or certain special symbols such as +, -, (.). It contains elementary language constructs such as variable names, labels, constants, key-words and operators. The compiler is to identify these types as classes. The lexical analyzer (scanner) takes as input (the source program) and separate the incoming text into pieces (tokens) such as constant, keywords, operators..... Etc.



Components of a Compiler:



A compiler must perform two major tasks: the analysis of a source program and the synthesis of its corresponding object program. The analysis task deals with the decomposition of the source program into its basic parts. Using these parts, the synthesis task builds their equivalent object program modules.

Lexical Analyzer (Scanner):

The source program is input to a lexical analyzer whose purpose is to separate the incoming text into pieces (tokens). Each class of token is given a unique internal representation number.

e.g.

- | | | |
|----------------------|--------|---|
| 1. variable name | —————> | 1 |
| 2. constant | —————> | 2 |
| 3. label | —————> | 3 |
| 4. addition operator | —————> | 4 |
| etc. | | |

Hence the statement:

test: if A>=B then x:=y;

<u>Token</u>	<u>rep. No.</u>
Test	3
:	12
If	20
A	1
>=	25
B	1
Then	21
X	1
:=	15
Y	1
;	10

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

Some scanners place constants, labels, keywords and variable names in appropriate tables. A table entry for a variable, for example may contain it's name, type (real, integer, Boolean) object program address, value and line in which it is declared.

Name	Type	Add.	Value	Line No.

The lexical analyzer supplies tokens to the syntax analyzer as a pair of items. The first is the address of that token in the symbol table and the second item is the representation number.

e.g.:

(1,3)	→	test
(7,21)	→	then

Syntax analyzer (parser):

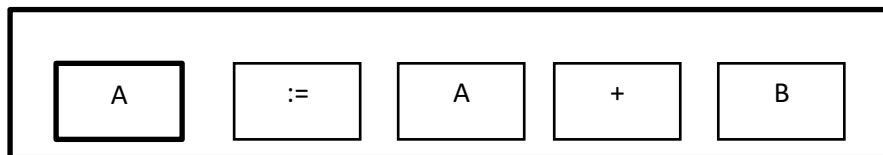
The syntax analyzer function is to take these tokens from scanner and determine the manner into which it is to be decomposed into constituent parts. It deals with the overall structure of the source program. The syntax analyzer groups the tokens into larger units (grammatical phrases) that are used by the compiler to synthesize output (e.g. / expression, statement, procedure function, etc)

Id:=id/cons./state.

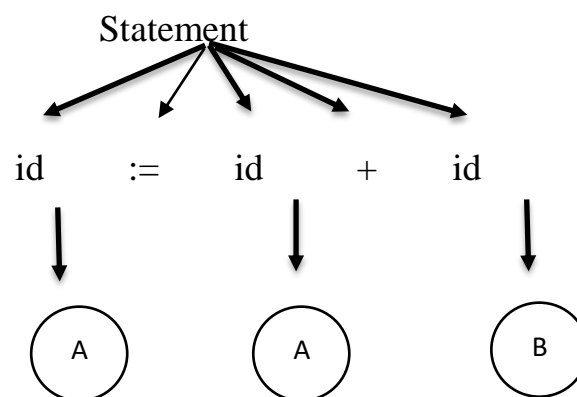
A=B ; (syntax error)

Advantage of the syntax analyzer is that all tokens are represented by fixed length information address (pointer) and integer.

e.g.:



The parser output a syntax tree (pars tree) the leaves of the tree are the tokens, and the non leaf nodes. Represent a syntactic class type.



Parse tree

Semantic analyzer:

The syntax tree is used by the semantic analyzer. The function of the semantic analyzer is to determine the meaning of the source program. Both analyzers work in close cooperation.

e.g.:

$(A+B) * (C+D)$

The semantic analyzer must determine what actions need to be performed when $*$, $+$, $/$, ...etc. is met, it checks whether both variables to be added have the same type (if not, the routine would probably make the same), and that both operands have values. The semantic analyzer interacts with the various tables of the compiler to perform its task.

The semantic analyzer generates intermediate form of the source code. For example $(A+B) * (C+D)$ generates the following quadruples: $(+, A, B, T_1)$, $(+, C, D, T_2)$, $(*T_1, T_2, T_3)$

Intermediate Code Generation:

After syntax & Semantic analysis, some compilers generate an explicit intermediate representation of the source program. We can think of this intermediate representation as a program for an abstract machine. This intermediate representation should have two import properties:

- 1) It should be easy to produce.
- 2) and easy to translate into the target program.

The intermediate representation can have a variety of forms:

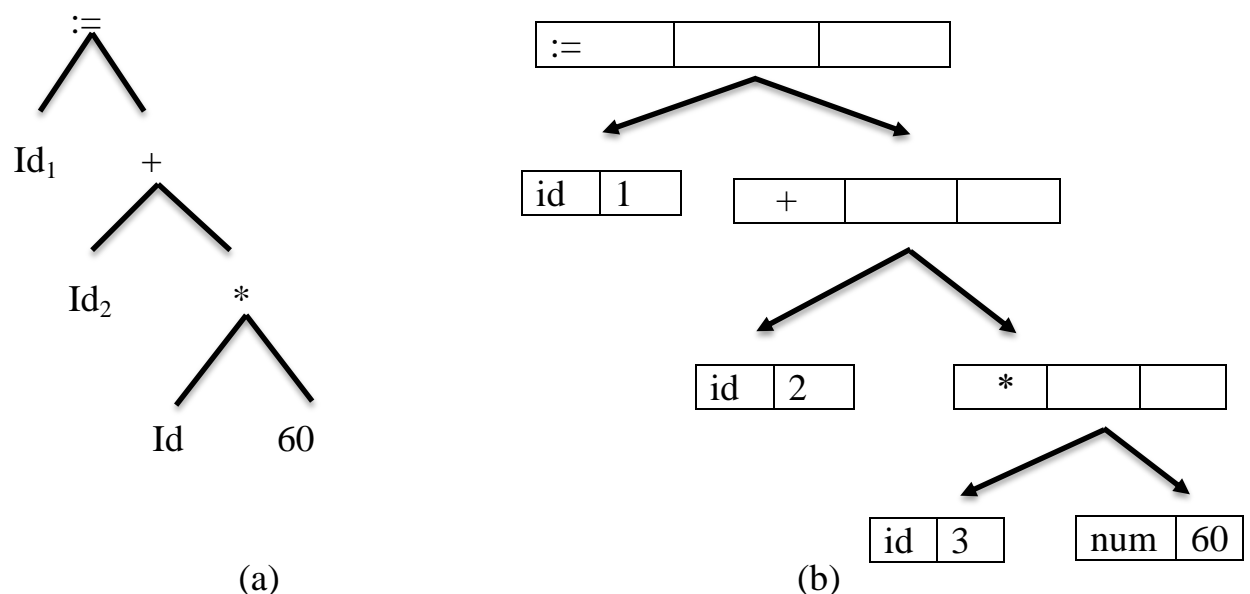
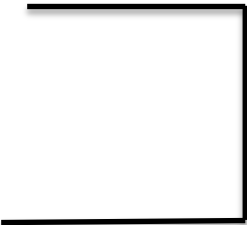


Fig. 2: the data structure in (b) is for the tree in (a)

Code optimization: the code optimization phase attempts to improve the intermediate code, so that faster running machine code will result. Some optimizations are trivial for example, a natural algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation using the two instruction (1.4):

Temp₁: = int to real (60)
 Temp₂: = id₃ * temp₁
 Temp₃: = id₂ + temp₂
 Id₁ := temp₃



----- (1.3)

Temp₁: = id₃ * 60.0
 id₁ := id₂ + temp₁




----- (1.4)

Code Generation: The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variable to registers.

For example, using register 1 & 2, the translation of the code of (id₁ = id₂+id₃*60) might become:

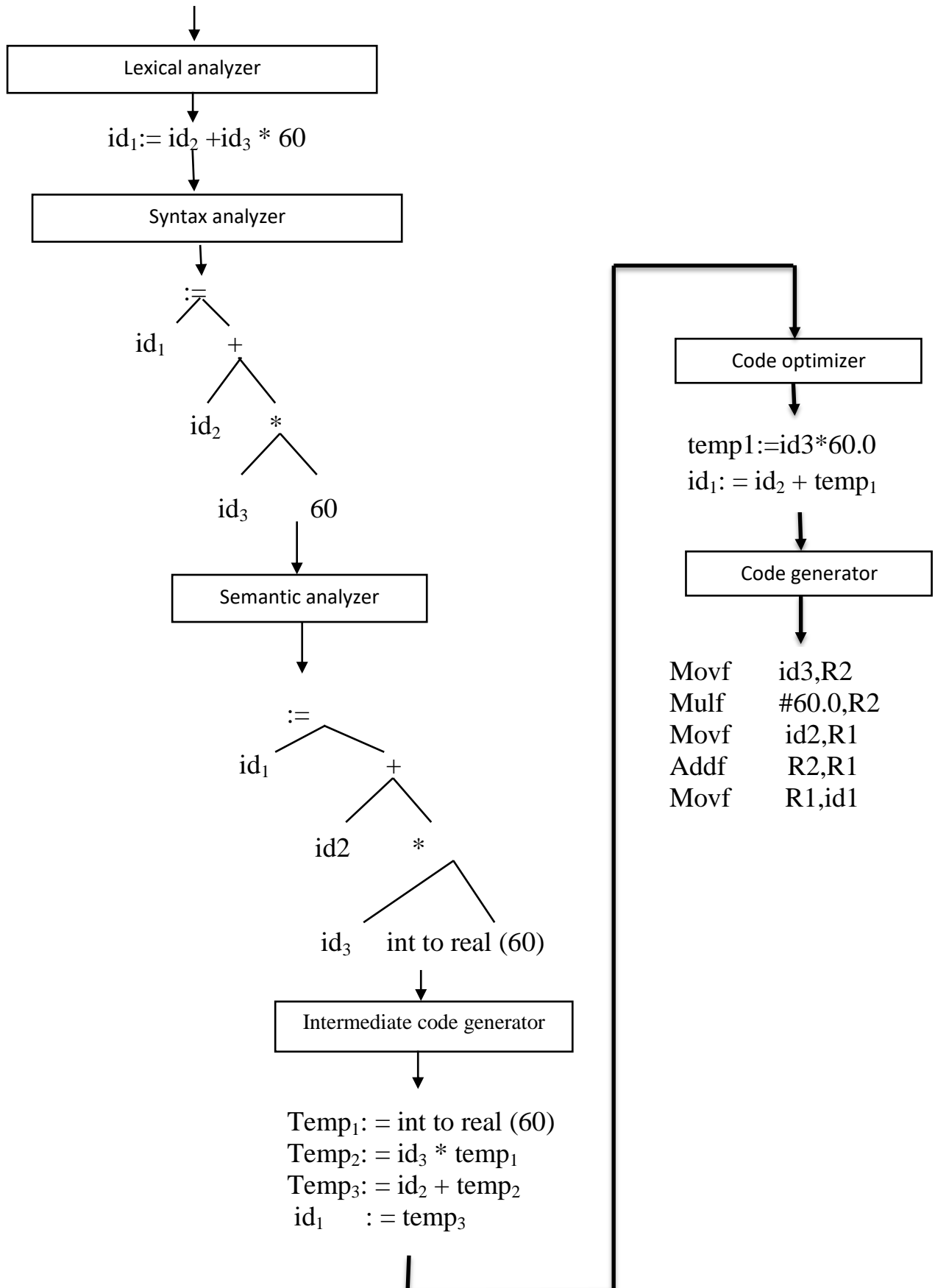
MovF id₃ , R₂
 MulF #60.0 , R₂ (#:constant)
 MovF id₂ , R₁
 ADDF R₂ , R₁
 MovF R₁ , id₁



Floating point source destination

The translation of a statement:

e.g. position: =initial + rate * 60

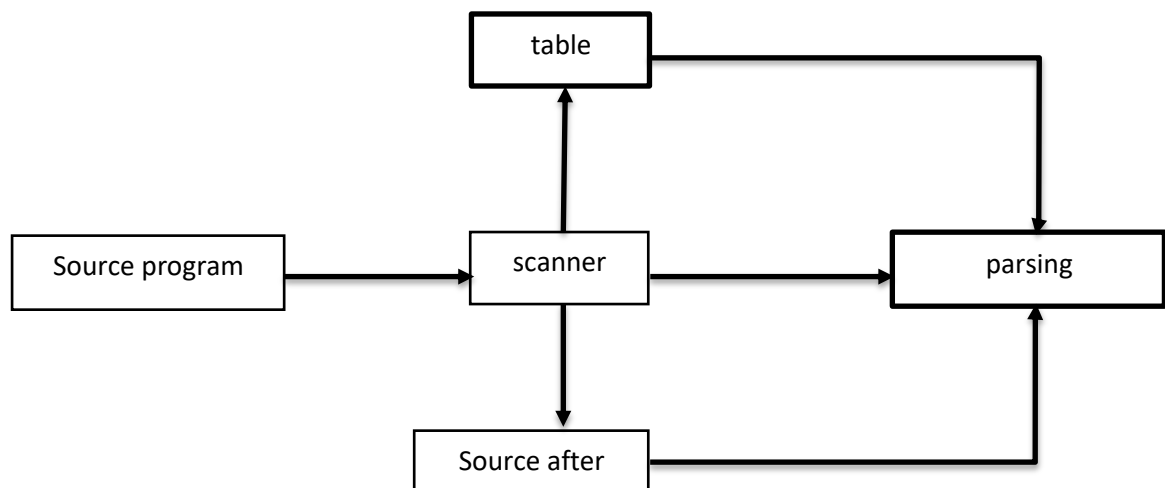


Scanners: the scanner represents an interface between the source program and the parser as we mentioned before the parser usually generates a syntax tree of a source program as defined by a grammar. The leaves of the tree are the terminal symbol of the grammar. It is these terminal symbols or tokens which the scanner extracts from the source code and passes the parser. It is possible for the parser to use the terminal chars. Set of the language as the set of tokens, but since tokens can be defined in term of simpler regular grammars rather than the source complex grammar used by the parser, it becomes desirable to use scanners. Using only parsers can become costly in terms of execution time and memory requirements, complexity and execution time can be reduced using a scanner.

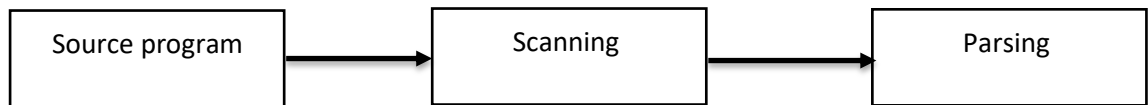
The separation of scanning and parsing can also have other advantages. It will make the process more efficient, and more information can be available to the parser when it is needed.

The scanner interacts with the parser in one of two ways:

- 1- The scanner may process the source program in a separate pass before parsing begins. Thus, the tokens are stored in a file or large table.



- 2- The way involves an iteration between the parser and the scanner. The scanner is called by the parser wherever the next token in the source program is required an interval from of complete source program does not need to be constructed and stored in memory before parsing can begin (hence it required less memory).



Token representation: token can be described in several ways. One way is by using regular grammar.

e.g.

<unsigned integer> : 0/1/2...../9/

0 <unsigned integer> /

1<unsigned integer> /

⋮

9<unsigned integer>

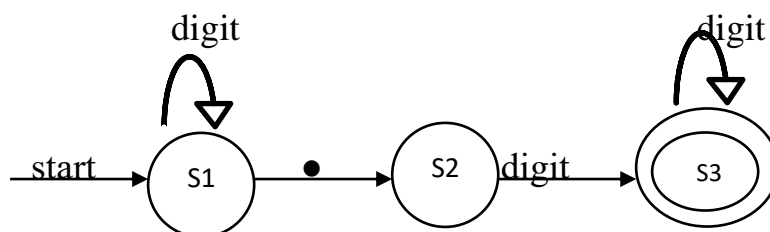
This grammar generates the set of natural numbers.

Describing tokens by means of how they can be recognized (or accepted) is done in terms of a mathematical model called a finite state acceptor (FSA).

FSA or finite automaton is a machine consisting of a read a head & finite state control box. The machine reads a tape one character at a time (from left to right).

There are finite numbers of state that the FSA can be in. It begins by the starting states, and if the acceptor read beyond the last character of the input string and the machine was in finite state then the string is accepted.

e.g.:

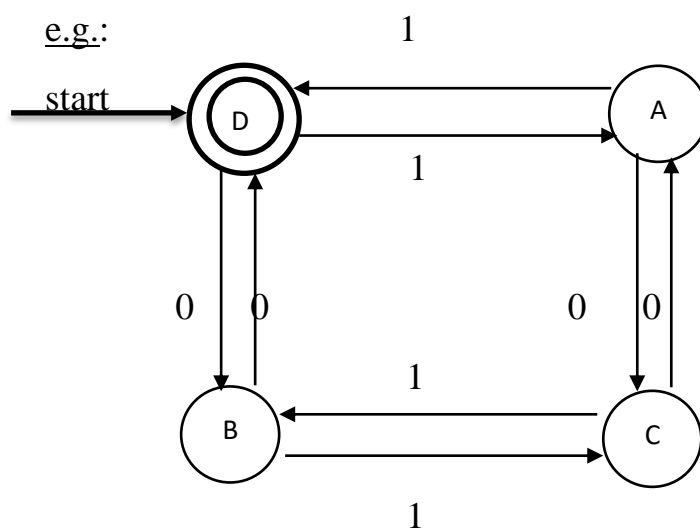


real numbers

Types of FSA:

- 1- Deterministic finite-state acceptor (DFA)
- 2- Non deterministic FSA (NFA).
- 3- Non deterministic FSA with Σ -transition

- 1- DFA: is an acceptor which for any state and input character has at most one transition state that acceptor changes to. If no transition state is specified the input string is rejected.



Acceptor which accepts string of even no. of 0's and even no. of 1's.

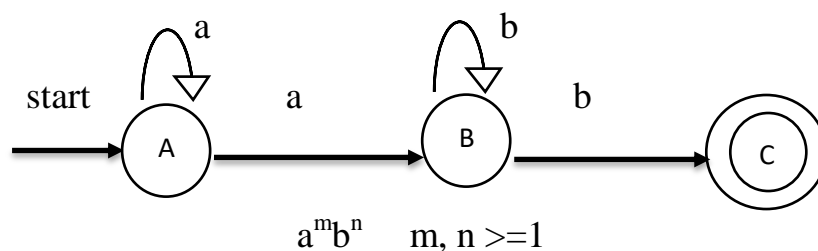
11 accept 1100 accept

00 accept 1010 accept

101 not accept 100 not accept 1101 not accept

- 2- NFA: similar to the DFA except that there may be more than one possible state attainable from a given stack for the same input char.

e.g.:

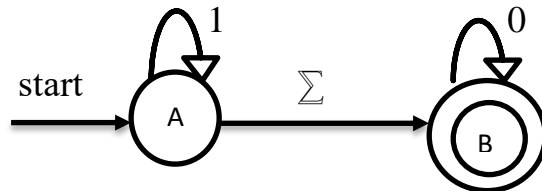


aa ab , abb accepted

aba not accepted

3- NFA with Σ -transition

similar to the NFA excepts that it accept the empty string as input.



Machine that accepts string of the form $1^m 0^n$ where $m, n \geq 0$

e.g.

Σ accept 1 accept 0 accept 1110 accept 0000 accept

101 not accept 01 not accept

Types of grammer :

A grammar is the $\langle V_n, V_t, S, P \rangle$

Where V_n : set of non termenal symbols

V_t : set of termenal symbols

S : starting symbols

P : production rules

Type 0 : (phrase- structure- grammer):

contain the productions of the form

$A \rightarrow \alpha$

e.g. $D \rightarrow a+b$ $a \wedge t \rightarrow W$ $S \rightarrow \epsilon$ $d \rightarrow f$ accepted

$\epsilon \rightarrow F$ not accept

Where:

A : is nonempty String of terminal and or non terminal symbols

α : is a string of terminal and /or non terminal symbols.

Type 1: (context _ sensitive grammer)

$\alpha \rightarrow \beta \quad |\alpha| \leq |\beta|, \quad (\beta \text{ not empty})$

Where:

α, β are strings of terminal and / or nonterminal

e.g. $D \rightarrow a+b$ $d \rightarrow F$ accepted
 $a^t \rightarrow W$ $S \rightarrow \epsilon$ $\epsilon \rightarrow F$ not accepted

Type 2 : (context _ Free grammar)

$$\alpha \rightarrow \beta \quad |\alpha| \leq |\beta|, \quad \alpha \in V_n$$

Left side of the grammar must be single non terminal.

e.g. $D \rightarrow a+b$ $G \rightarrow A$ accept

$d \rightarrow F$ $a^t \rightarrow W$ $S \rightarrow \epsilon$ $\epsilon \rightarrow F$ not accepted

Type 3 : (Regular grammar)

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

$\alpha \in V_n$, β has the form of a or aB

where $a \in V_T$ and $B \in V_n$

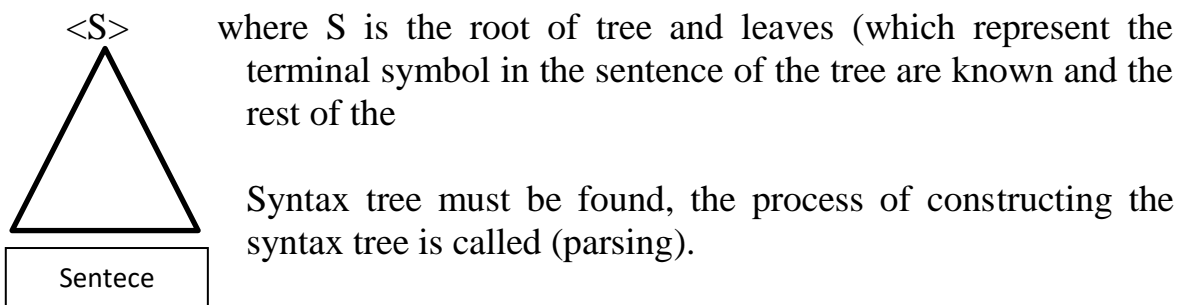
e.g. $D \rightarrow a+b$ $H \rightarrow d*F$ $L \rightarrow e$ accepted
 $G \rightarrow A$ $d \rightarrow F$ $a^t \rightarrow W$ $S \rightarrow \epsilon$ $\epsilon \rightarrow F$ not accepted

Syntax analysis (parsing):

Every programming language has rules that describe the syntactic structure of well formed programs. In Pascal for example, program is made out of the blocks, a block out of statement; a statement out of expressions, an expression out of tokens and so on.

The syntax of programming language constructs can be described by context Free grammar or BNF notation.

Given sentence in the language, the construction of a parser can be illustrated as follows :



Types of parsing :

- 1- Top-down parsing : start from the root and move down wards the leaves.

- 2- Bottom-up parsing : start at the leaves (the input string or sentence) and moving upward towards the root.
-

Top-down parsing: is characterized by presenting three methods of performing such parses:

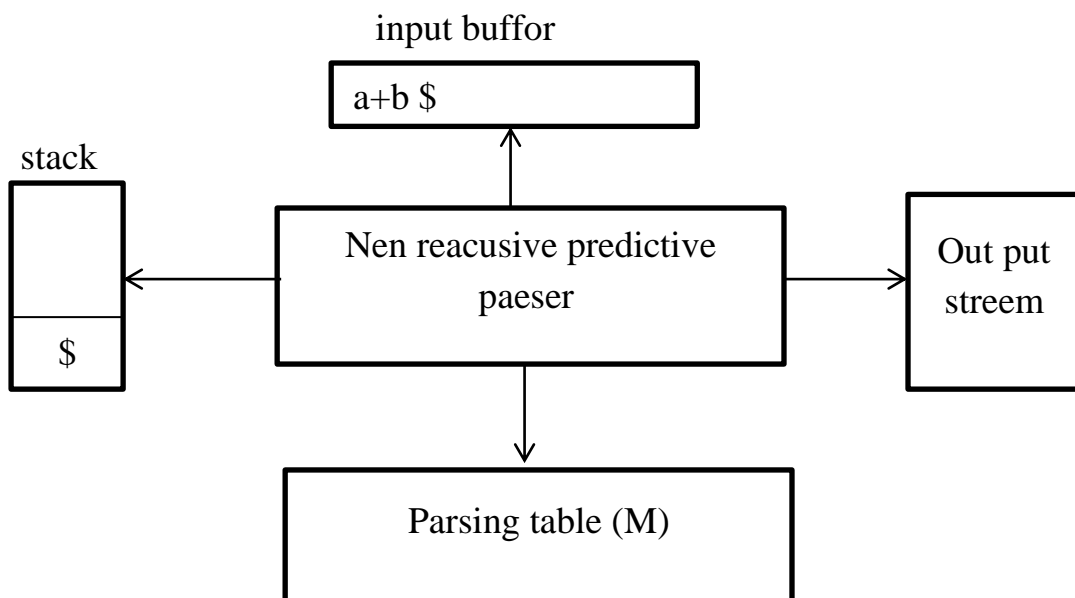
- a) Brute-Force method (full-back tracking)
- b) Recursive-Decent method (with-no back tracking)
- c) Parsing with limited back tracking

Top-down parsing

Non-Recursive predictive parser:

It is possible to write a predictive parser using a stack, rather than recursive calls.

The non-recursive parser looks up a production to be applied in a parsing table . The parser has an input buffer, stack, a parsing table and the output stream.



The input buffer contains the string to be parsed followed by \$ (right most end marker).

The stack contains a sequence of a grammar symbols with \$ (on the Bottom of stack).

Initially stack contains **S\$** where S is the start symbol of grammar .

The parsing table is 2-dimensional array $M[A,a]$ where “A” is a non terminal symbol and “a” is a terminal or \$ contains the production table applied.

The output string consists of production rules that have been applied.

The parser behaves as follows:

Let x be the top element of stack, let a be the current input symbol,

- if $x=a=\$$ then a successful parse.
- if $x=a \neq \$$ then pop the top element of stack and advance the input pointer to the next input symbol.
- if x is a non terminal then consult the entry $M[x,a]$ of the parsing table M .

If the entry contains an x production of the form $X \rightarrow UVW$, then the parser replace X by WUV where U on top.

However, the $M[X,a]$ entry might not contain an X production so there is an error and the parser calls the required error routine to recover.

The algorithm :

A string W and parsing table M for grammar G are input, if W is in $L(G)$, then the left most derivation of W is output, otherwise an error indication.

Set ip to point to first symbol of w
repeat

Let X be the top symbol of stack and a be the symbol pointed to by ip

If $X=a$ then pop X from stack and advance ip

Else Error () /*not match terminal*/

Else /* X is non terminal */

If $M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_K$ then

begin

pop X from top of stack

push $Y_K Y_{K-1} Y_{K-2} \dots Y_2 Y_1$ on to stack with y_1 on top

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

end

else error ()

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

Example: Consider the following grammar ;

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

and the following parsing table M for the grammar

Non terminal	Input symbol					
	I	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow + TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow i$			$F \rightarrow (E)$		

suppose input string $i+i*i$, then give the steps of predictive parsing.

Sol. :

Stack	In put buffer	Out put
\$ E	i +i*i \$	
\$ E' T	i +i*i \$	$E \rightarrow TE'$
\$ E' T' F	i +i*i \$	$T \rightarrow FT'$
\$ E' T' i	i +i*i \$	$F \rightarrow i$
\$ E' T'	+i*i \$	Pop i
\$ E'	+i*i \$	$T' \rightarrow \epsilon$
\$ E' T +	+i*i \$	$E' \rightarrow + TE'$
\$ E' T	i*i \$	Pop +
\$ E' T' F	i*i \$	$T \rightarrow FT'$
\$ E' T' i	i*i \$	$F \rightarrow i$
\$ E' T'	*i \$	Pop i
\$ E' T' F*	*i \$	$T' \rightarrow *FT'$
\$ E' T' F	i \$	Pop *
\$ E' T' i	i \$	$F \rightarrow i$
\$ E' T'	\$	Pop i
\$ E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

Successful parsing

First and Follow:

The construction of a predictive parser is aided by two functions associated with a grammar G . These functions are **first** and **follow**, allow us to fill in the entries of predictive parsing table for G . Wherever possible, let α be any string of grammar symbols of G , $\text{First}(\alpha)$ is the set of all terminals that begins the strings derived from α .

e.g:

$$1- A \rightarrow aABC$$

$$\text{First}(aABC) = \{a\}$$

$$2- A \rightarrow \epsilon$$

$$\text{First}(A) = \{\epsilon\} \quad \epsilon \text{ is in first of } A$$

$$3- A \rightarrow B$$

$$B \rightarrow \epsilon$$

$$\text{First}(A) = \{\epsilon\}$$

Let A be a nonterminal of G , **follow**(A) is the set of all terminals that follows A .

i.e.

The set of all terminals that can appear immediately to the right of A in some sentential form, that the set of all a 's such that there exist a derivation

Ex:

$S \rightarrow \alpha A a \beta$ for some α and β which are string of terminals and nonterminals.

$$\text{follow}(A) = \{a\}$$

e.g:

$$A \rightarrow aBcCd$$

$$\text{Follow}(B) = \{c\}$$

$$\text{Follow}(C) = \{d\}$$

Note:-

- During the derivation there may have been symbols between A and a , and if they do so then they derived ϵ and disappear.

$$\text{e.g.} \quad S \rightarrow bACa \quad C \rightarrow \epsilon \quad \text{follow}(A) = \{a\}$$

- If A can be the right most symbol of some sentential form then (\$) is in follow (A)

Note that \$ is right most end marker not a grammar symbol.

e.g:

$$S \rightarrow \alpha B \rightarrow \alpha abA$$

Follow (A) = { \$ }

Follow (B) = { \$ }

e.g.:

B \rightarrow abA

A \rightarrow EC / ϵ

E \rightarrow dF / ed

First (B) = { a }

First (A) = { d, e, c }

First (E) = { d, e }

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

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

e.g.

FIRST (a)={ a }

FIRST (b) = { b }

FIRST (c) = { c }

- 2- If $X \rightarrow \epsilon$ is a production , then add ϵ to first (x).

e.g.

$X \rightarrow abc/\epsilon$

FIRST (X) = { a , ϵ }

- 3- If X is nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_K$ is a production, then place **a** in FIRST (X) if for some **i** , **a** is in FIRST (Y_i) and ϵ is in all of FIRST (Y₁), FIRST (Y₂) FIRST(Y_{i-1})

e.g.

$X \rightarrow Y_1 Y_2 \dots Y_{i-1} Y_i \dots Y_k$

$Y_1 \rightarrow c / \epsilon$

$Y_2 \rightarrow b / \epsilon$

.

.

.

.

.

.

$Y_{i-1} \rightarrow r / \epsilon$

$Y_i \rightarrow a$

➤ To compute follow (X) for all nonterminals X, apply the following until nothing can be added to any follow set:

- 1- Place \$ in follow (S), where S is the start symbol and \$ is input right end marker.
- 2- If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST (β) except for ϵ is placed Follow (B).

كل ماموجود في First (β) ماعدا ϵ يوضع في Follow(B)

- 3- If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ (i.e. $\beta \rightarrow \epsilon$) then everything in Follow(A) is in Follow(B) .

كل ماموجود في Follow (A) يوضع في Follow(B)

e.g.

$S \rightarrow AE$

$A \rightarrow c / \epsilon$

$E \rightarrow bS / \epsilon$

Follow (S) = { \$ }

Follow (A) = { b , \$ }

Follow(E) = { \$ }

$S \rightarrow \underline{A}E \rightarrow c \underline{E} \rightarrow c bS$

$S \rightarrow \underline{A}\underline{E} \rightarrow A bS$

$S \rightarrow \underline{A}E$

Also remember that Follow (A) = { w / w $\in V_T$ and $S \rightarrow \alpha Aw\beta$ } for some α, β where S is start symbol.

Example : Consider the grammar

$E \rightarrow TE^{\setminus}$

$E^{\setminus} \rightarrow + TE^{\setminus} \setminus \epsilon$

$T \rightarrow FT^{\setminus}$

$T^{\setminus} \rightarrow * FT^{\setminus} \setminus \epsilon$

$F \rightarrow (E) / i$

Compute functions First & Follow for the grammar.

Sol.

First (E) = { (, i }

First (E[\]) = { + , ϵ }

First (T) = { (, i }

First (T[\]) = { * , ϵ }

First (F) = { (, i }

Follow (E) = { \$,) }

$\underline{E} \rightarrow \underline{T}E^{\setminus} \rightarrow \underline{F}T^{\setminus}E^{\setminus} \rightarrow (E) T^{\setminus}E^{\setminus}$

Follow (E[\]) = { \$,) }

$$E \rightarrow \underline{T}E' \rightarrow \underline{F}T'E' \rightarrow (\underline{E})T'E' \rightarrow (TE')T'E'$$

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

$$E \rightarrow T \underline{E}' \rightarrow T \epsilon$$

$$E \rightarrow \underline{T}E' \rightarrow \underline{F}T'E' \rightarrow (\underline{E})T'E' \rightarrow (TE')T'E' \rightarrow (T\epsilon)T'E'$$

$$E \rightarrow TE' \rightarrow T + TE'$$

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

H.W.

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

H.W.

Construction of predictive parsing tables:

Input: Grammar G .

Output: parsing table M .

Method:

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

Non terminal	Input symbols					
	I	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow i$			$F \rightarrow (E)$		

Example: Consider the following grammar

$$S' \rightarrow A\#$$

$$A \rightarrow i B \leftarrow e$$

$$B \rightarrow SB / \epsilon$$

$$S \rightarrow [e C] / \bullet i$$

$$C \rightarrow eC / \epsilon$$

Construct the parsing table for a predictive parser and give a trace of the stack contents when parsing the string: $i[e] \leftarrow e\#$

SOL.:

$$\text{First}(S') = \{i\}$$

$\text{First}(A) = \{i\}$
 $\text{First}(B) = \{[, \bullet, \epsilon\}$
 $\text{First}(S) = \{[, \bullet\}$
 $\text{First}(C) = \{e, \epsilon\}$

$\text{Follow}(S') = \{\$ \}$ S is start symbol

$\text{Follow}(A) = \{\#\}$

$S' \rightarrow A\#$

$\text{Follow}(B) = \{\leftarrow\}$

$S' \rightarrow A\# \rightarrow iB \leftarrow e\#$

$\text{Follow}(S) = \{\leftarrow, [, \bullet\}$

$S' \rightarrow \underline{A}\# \rightarrow i\underline{B} \leftarrow e\# \rightarrow i\underline{SB} \leftarrow e\# \rightarrow i\underline{SSB} \leftarrow e\# \rightarrow i\underline{SS} \leftarrow e\#$
 $\rightarrow iS[eC] \leftarrow e\#$

$\rightarrow iS \bullet i \leftarrow e\#$

$\text{Follow}(C) = \{\}$ H.W. drive that

$S' \rightarrow \underline{A}\# \rightarrow i\underline{B} \leftarrow e\# \rightarrow i\underline{SB} \leftarrow e\# \rightarrow i\underline{SSB} \leftarrow e\# \rightarrow iSB \leftarrow e\# \rightarrow$
 $iS[eC] \leftarrow e\#$

Non terminal	Input symbols							
	i	e	•	[]	←	#	\$
S'	$S' \rightarrow A\#$							
A	$A \rightarrow iB \leftarrow e$							
B			$B \rightarrow SB$	$B \rightarrow SB$		$B \rightarrow \epsilon$		
S			$S \rightarrow \bullet i$	$S \rightarrow [eC]$				
C		$C \rightarrow eC$			$C \rightarrow \epsilon$			

Stack	Input buffer	output
$\$S'$	$i[e] \leftarrow e\#\$$	
$\$ \# A$	$i[e] \leftarrow e\#\$$	$S' \rightarrow A\#$
$\$ \# e \leftarrow B i$	$i[e] \leftarrow e\#\$$	$A \rightarrow iB \leftarrow e$
$\$ \# e \leftarrow B$	$[e] \leftarrow e\#\$$	Pop i
$\$ \# e \leftarrow B S$	$[e] \leftarrow e\#\$$	$B \rightarrow SB$
$\$ \# e \leftarrow B] C e [$	$[e] \leftarrow e\#\$$	$S \rightarrow [eC]$
$\$ \# e \leftarrow B] C e$	$e] \leftarrow e\#\$$	Pop [
$\$ \# e \leftarrow B] C$	$] \leftarrow e\#\$$	Pop e
$\$ \# e \leftarrow B]$	$] \leftarrow e\#\$$	$C \rightarrow \epsilon$
$\$ \# e \leftarrow B$	$\leftarrow e\#\$$	Pop]
$\$ \# e \leftarrow$	$\leftarrow e\#\$$	$B \rightarrow \epsilon$
$\$ \# e$	$e\#\$$	Pop ←

\$#
`\$

#\$
\$

Pop e
Pop #

Ssuccessful parse

Example: Consider the following grammar:

$$S \rightarrow iEtSS^{\backslash} / a$$

$$S^{\backslash} \rightarrow eS / \epsilon$$

$$E \rightarrow b$$

Construct a parsing table for a predictive parser.

SOL.:

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

$$\text{First}(S^{\backslash}) = \{e, \epsilon\}$$

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

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

$$\text{Follow}(S^{\backslash}) = \{e, \$\}$$

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

Non terminal	Input symbol					
	i	t	a	e	b	\$
S	$S \rightarrow iEtSS^{\backslash}$		$S \rightarrow a$			
S^{\backslash}				$S^{\backslash} \rightarrow eS$ $S^{\backslash} \rightarrow \epsilon$		$S^{\backslash} \rightarrow \epsilon$
E					$E \rightarrow b$	

Multiple defined

The entry for $M[S^{\backslash}, e]$ contains both $S^{\backslash} \rightarrow eS$ and $S^{\backslash} \rightarrow \epsilon$, since

$$\text{Follow}(S^{\backslash}) = \{e, \$\}$$

The grammar is ambiguous (not predictive) and the ambiguity is manifested by a choice in what production to use when an e is seen.

We can remove the ambiguity if we choose $S^{\backslash} \rightarrow eS$. Note that the choice $S^{\backslash} \rightarrow \epsilon$ would prevent (e) from ever being put on the stack or removed from the input, and therefore surely wrong.

A grammar whose parsing table has no multiply defined entries is said to be LL(1) grammar. The first L stands for scanning the input from left to right. The second L for producing a left most derivation & the "1" for using one input symbol of look ahead at each step to make parsing decision.

LL (1) grammar has properties:

- 1- No ambiguous grammar.
- 2- No left recursive grammar.
- 3- No left factoring grammar.

It can also be shown that a grammar G is LL(1) if whenever $A \rightarrow \alpha / \beta$ are two distinct productions of G the following conditions hold:

- 1- For no terminal a do both α and β derive strings beginning with a .

$$\text{first}(\alpha) \cap \text{first}(\beta) = \emptyset$$
- 2- At most one of α or β can derive the empty string.

$$\alpha \rightarrow \epsilon \text{ or } \beta \rightarrow \epsilon$$
- 3- If β indirectly $\beta \rightarrow \epsilon$ then α does not derive any string beginning with a terminal in follow (A).

$$\text{First}(\alpha) \cap \text{Follow}(A) = \emptyset$$

Ex: Verify if the following grammar is LL(1) grammar:

$A \rightarrow Bb / Cd$

$B \rightarrow aB / \epsilon$

$C \rightarrow cC / \epsilon$

- 1- Parsing table
- 2- Give a trace of aab , ccd

Sol :-

$\text{First}(A) = \{ a, b, c, d \}$

$\text{First}(B) = \{ a, \epsilon \}$

$\text{First}(C) = \{ c, \epsilon \}$

$\text{Follow}(A) = \{ \$ \}$

$\text{Follow}(B) = \{ b \}$

$\text{Follow}(C) = \{ d \}$

Condition1 : $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$

$\text{First}(Bb) \cap \text{First}(Cd)$
 $\{a, b\} \cap \{c, d\} = \emptyset$
 $\text{First}(aB) \cap \text{First}(\epsilon) = \emptyset$
 $\{a\} \cap \{\epsilon\} = \emptyset$
 $\text{First}(cC) \cap \text{First}(\epsilon) = \emptyset$
 $\{c\} \cap \{\epsilon\} = \emptyset$
 Condition 1 is O.K.

Condition2 :
 Production 1: $\alpha \neq \epsilon \ \& \ \beta \neq \epsilon$
 Production 2 $\beta \rightarrow \epsilon$
 Production 3 $\beta \rightarrow \epsilon$
 Condition2 is O.K.

Condition3 : If $\beta \rightarrow \epsilon$ then $\text{First}(\alpha) \cap \text{Follow}(A) = \emptyset$
 1- $\beta \neq \epsilon$
 2- $\beta \rightarrow \epsilon$ then $\text{First}(aB) \cap \text{Follow}(B) = \emptyset$
 $\{a\} \cap \{b\} = \emptyset$
 3- $\beta \rightarrow \epsilon$ then $\text{First}(cC) \cap \text{Follow}(C) = \emptyset$
 $\{c\} \cap \{d\} = \emptyset$
 Condition3 is O.K.

The grammar is LL(1).

Ambiguous Grammars

Ambiguous Grammar: Given a string there are two or more left derivations (or right derivation), or two or more derivation trees.
Unambiguous Grammar: Given a string there is exactly one derivation tree.

It is theorem that every ambiguous grammar fails to be LL(1).

Ex.1 : (Ambiguous Grammars)

- 1- $E \rightarrow E + E$
- 2- $E \rightarrow E * E$
- 3- $E \rightarrow (E)$
- 4- $E \rightarrow -E$
- 5- $E \rightarrow \text{id}$

Consider the string : $\text{id} + \text{id} * \text{id}$

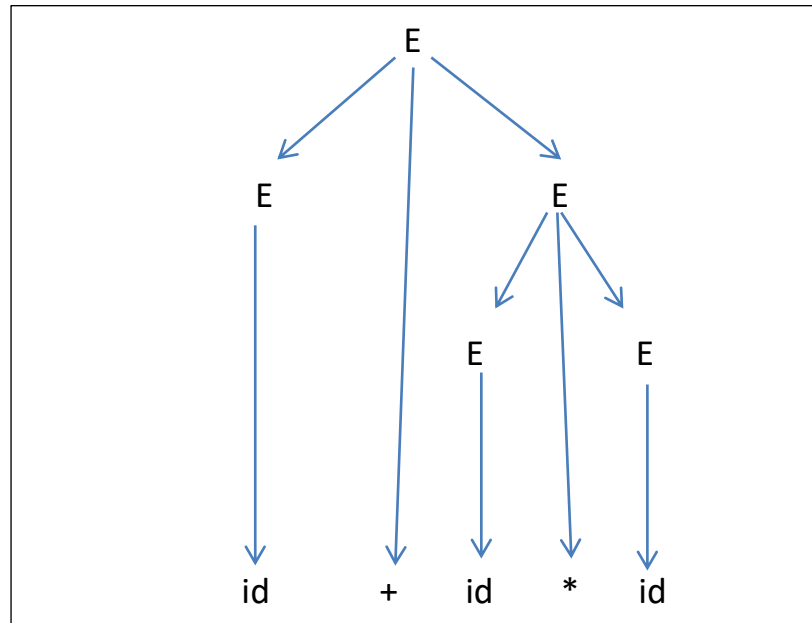
Derivation 1 :

$E \rightarrow \underline{E} + E \quad (1)$

$E \rightarrow id + \underline{E} \quad (5)$
 $E \rightarrow id + \underline{E} * E \quad (2)$
 $E \rightarrow id + id * \underline{E} \quad (5)$
 $E \rightarrow id + id * id \quad (5)$

Derivation tree1

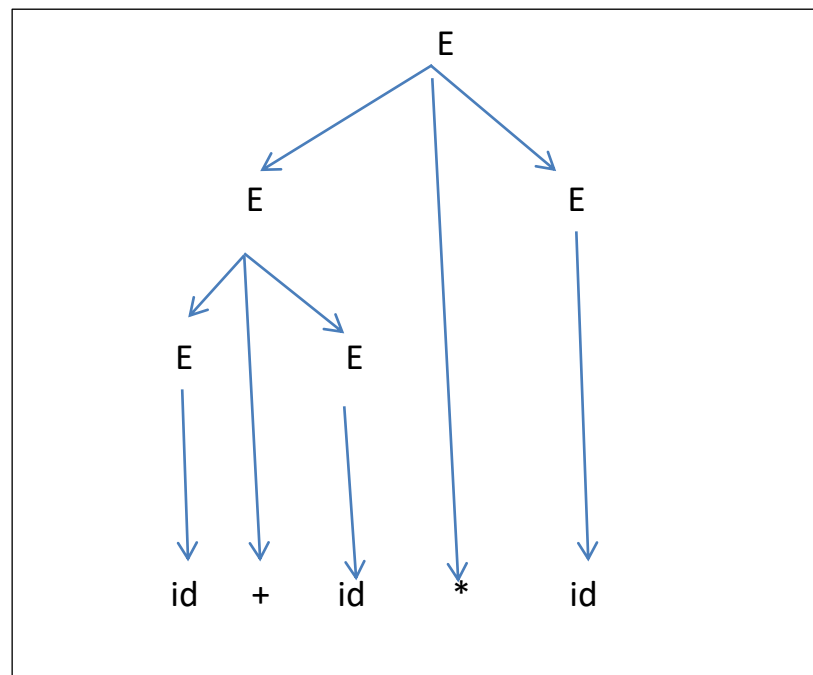
E



Derivation 2 :

$E \rightarrow \underline{E} * E \quad (2)$
 $\rightarrow \underline{E} + E * E \quad (1)$
 $\rightarrow id + \underline{E} * E \quad (5)$
 $\rightarrow id + id * \underline{E} \quad (5)$
 $\rightarrow id + id * id \quad (5)$

Derivation tree2



Example 2 :- (unambiguous Grammar)

1- $E \rightarrow E + T$

2- $E \rightarrow T$

3- $T \rightarrow T + F$

4- $T \rightarrow F$

5- $F \rightarrow (E)$

6- $F \rightarrow id$

String : $id + id * id$

There is only one possible derivation

Derivation :-

$E \xrightarrow{1} E + T$

$\xrightarrow{2} T + T$

$\xrightarrow{4} F + T$

$\xrightarrow{6} id + T$

$\xrightarrow{3} id + T * F$

$\xrightarrow{4} id + F * F$

$\xrightarrow{6} id + id * F$

$\xrightarrow{6} id + id * id$

Ex 3 :- (Ambi . Gra. " Dangling Else "

1- $S \rightarrow \text{if } c \text{ then } S$

2- $S \rightarrow \text{if } c \text{ then } S1 \text{ Else } S$

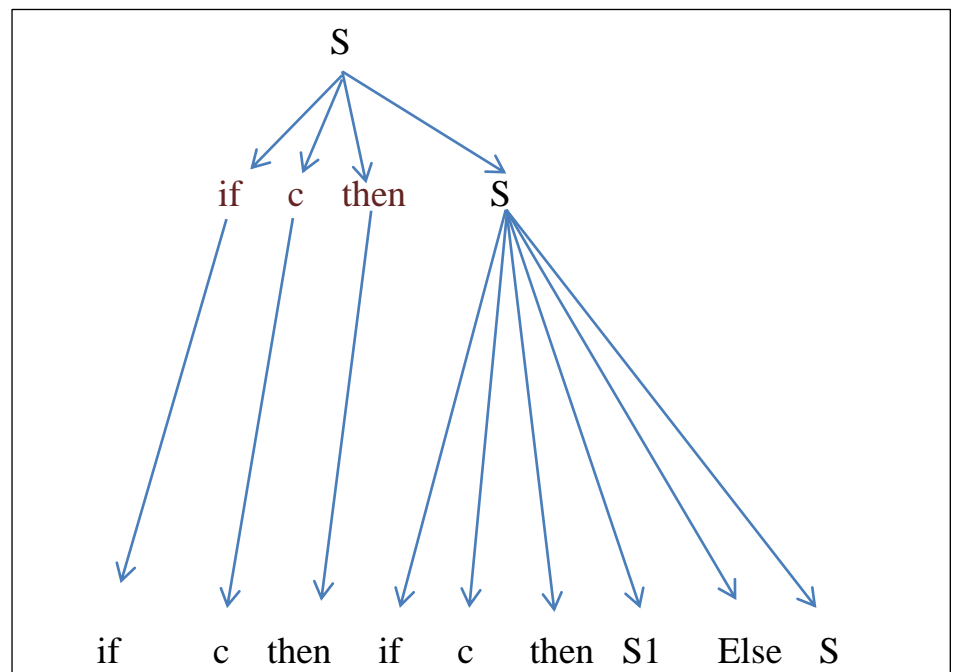
3- $S1 \rightarrow \text{if } c \text{ then } S1$

Consider the string is : **if c then if c then S1 Else S**

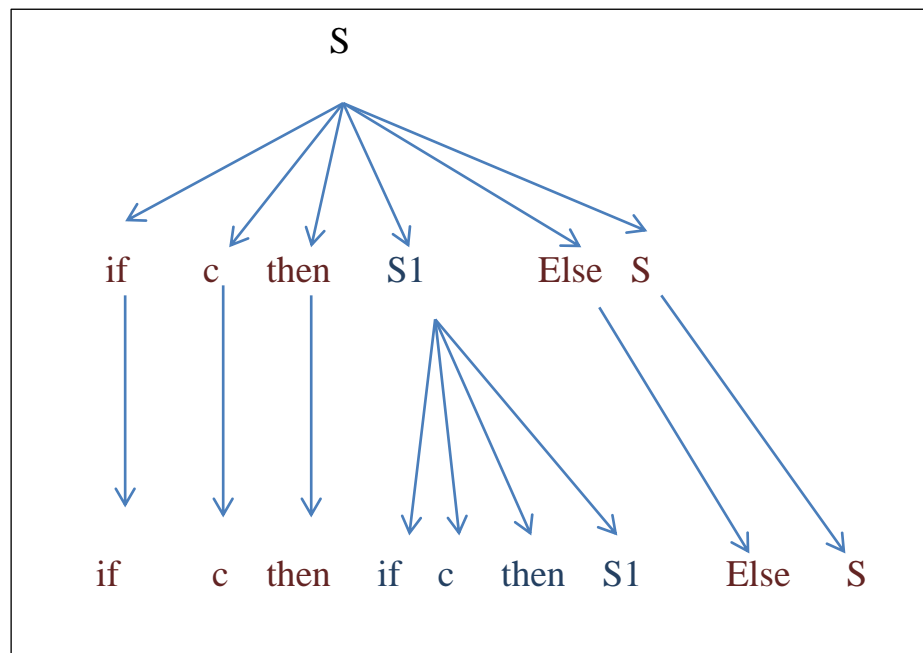
deriv.1:

$S \xrightarrow{1} \text{if } c \text{ then } \underline{S} \xrightarrow{2} \text{if } c \text{ then if } c \text{ then } S1 \text{ Else } S$

Derivation tree1:



deriv.2: $S^2 \rightarrow \text{if } c \text{ then } \underline{S1} \text{ Else } S^3 \rightarrow \text{If } c \text{ then if } c \text{ then } S1 \text{ Else } S$
 Derivation tree2 :



Ex 4 :- (unamb. gra. for if then-else)

1- $S \rightarrow \text{if } c \text{ then } S$

2- $S \rightarrow \text{if } c \text{ then } S1 \text{ else } S$

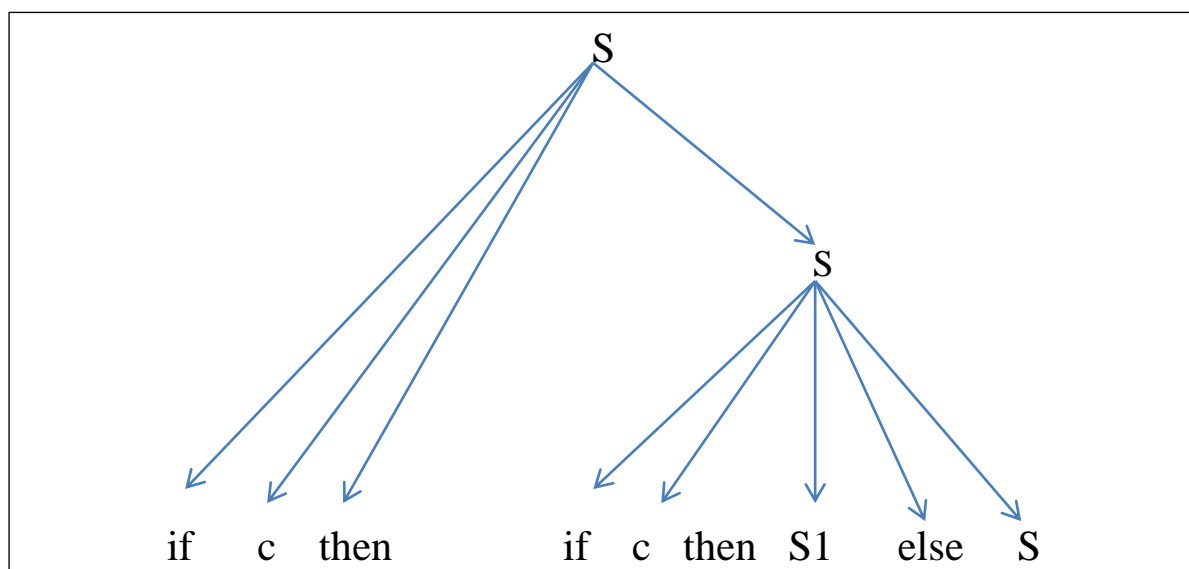
3- $S1 \rightarrow \text{if } c \text{ then } S1 \text{ else } S$

The string :- **if c then if c then S1 else S**

Derivation :-

$S \rightarrow \text{if } c \text{ then } S$

$\rightarrow \text{if } c \text{ then if } c \text{ then } S1 \text{ else } S$



Left Recursion :- الاستدعاء الذاتي

A non terminal is called left recursive if a string beginning with that non terminal can be derived from that non terminal itself by applying one or more production .

This infinite loop can be avoided by eliminating left recursion which involved performing the following:

Let the rule of the left recursion non terminal A be:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

Where β_i does not begin with A, then replace this rule by:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \epsilon$$

e.g $S \rightarrow aAc$

$$A \rightarrow Ab \mid \epsilon$$

Sol. $A \rightarrow A'$
 $A' \rightarrow bA' \mid \epsilon$

Example:

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

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

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

Sol.

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Ex: Consider the grammar:

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L, S \mid S$$

Can a predictive parser be constructed for the above grammar?

Sol.: since the above grammar has Left – Recursive rule

It is not LL(1) grammar then to eliminate left recursion:

$$1- S \rightarrow (L) \mid a$$

$$2- L \rightarrow SL'$$

$$3- L' \rightarrow , SL' \mid \epsilon$$

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

$$\text{First}(L) = \{ (, a \}$$

$$\text{First}(L') = \{ ', \epsilon \}$$

Follow (S) = { ' ,) , \$ }

Follow (L) = {) }

Follow (L') = {) }

Cond.1 : $\text{first}(\alpha) \cap \text{first}(\beta) = \emptyset$

1- $\text{first}((L)) \cap \text{first}(a) = \emptyset$

$$\{() \cap \{a\} = \emptyset$$

3- $\text{first}(, SL') \cap \text{first}(\epsilon) = \emptyset$

$$\{ , \} \cap \{ \epsilon \} = \emptyset$$

Cond.1 is o.k.

Cond. 2 : $\alpha \rightarrow \epsilon$ or $\beta \rightarrow \epsilon$

1- not α nor β , $\rightarrow \epsilon$

2- not α nor β , $\rightarrow \epsilon$

3- $\beta \rightarrow \epsilon$

Cond.2 is o.k

Cond 3 :- if $\beta \rightarrow \epsilon$ then $\text{First}(\alpha) \cap \text{follow}(A) = \emptyset$

3- $\text{first}(, SL') \cap \text{follow}(L') = \emptyset$

$$\{ , \} \cap \{) \} = \emptyset$$

Cond.3 is o.k

The grammar is LL1.

	a	,	()	\$
S	$S \rightarrow a$		$S \rightarrow (L)$		
L	$L \rightarrow S L'$		$L \rightarrow S L'$		
L'		$L' \rightarrow , S L'$		$L' \rightarrow \epsilon$	

Left Factoring

Left Factoring: It is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. when it is not clear which of two productions to use to expand a nonterminal **A**, we may be able to rewrite the **A** productions to defer the decision until we have seen enough of the input to make the right choice.

ملاحظة :- مع كل α يجب ان يوجد β واذا لم نجد نضع ϵ .

The method is as follows:

If $A \rightarrow \alpha \beta_1 / \alpha \beta_2 / \alpha \beta_3 \dots / \alpha \beta_n / \epsilon$

When \emptyset represents all alternatives that doesn't begin with α

Then if $\alpha \neq \epsilon$ replace all the A production by

$$A \rightarrow \alpha A' \emptyset$$

$$A' \rightarrow \beta_1 / \beta_2 / \dots / \beta_n$$

e.g. :

1. st-seq \rightarrow stmt ; st-seq / stmt
2. stmt \rightarrow s

Sol :

1. st-seq \rightarrow stmt st-seq'
2. st-seq' \rightarrow ; st-seq \ ϵ
3. stmt \rightarrow s

e.g:

$$\text{exp} \rightarrow \text{term} + \text{exp} / \text{term}$$

sol :

$$\begin{aligned} \text{exp} &\rightarrow \text{term exp}' \\ \text{exp}' &\rightarrow + \text{exp} / \epsilon \end{aligned}$$

Error Recovery in Predictive Parsing

The following remarks helps in the recovery:

- 1- Place all symbols in Follow (A) as the synchronizing set for A. If we skip tokens until an element of follow(A) is seen and pop A from the stack, it is likely that parsing can continued.
- 2- Add symbols in First (A) to the synchronizing set for A, that it is possible to resume parsing according to A if a symbol in first A appears in the input.
- 3- Add to the synchronizing set the symbols that begin higher constructs of the grammar (e.g. Keywords).
- 4- If a nonterminal can generate the empty string, the production derive ϵ can be used as a default.
- 5- If a terminal on top of stack can not be matched then issue a message, pop the terminal of the stack, and continue parsing.

Example : consider the grammar:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) / i$$

First (E) = First (T) = First (F) = { (, i }

First (E') = { + , € }

First (T') = { * , € }

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

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

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

Non terminal	In put symbol					
	i	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow i$	synch	synch	$F \rightarrow (E)$	synch	synch

The table can be consulted during parsing such that:

1. If the parser looks an entry M [A ,a] and it is **blank** , then the input symbol " a " is skipped.
2. If the entry is " synch " , then the nonterminal on top of the stack is popped.
3. If a token on top of stack doesn't match the input symbol then pop the token from stack.

e.g. Given the following string which is an error:

+ i * + i

The parser can recover as follows

Stack	Input buffer	Out put
\$ E	+ i * + i \$	blank , skip +
\$ E	i * + i \$	$E \rightarrow TE'$
\$ E' T	i * + i \$	$T \rightarrow FT'$
\$ E' T' F	i * + i \$	$F \rightarrow i$
\$ E' T' i	i * + i \$	Pop i
\$ E' T'	* + i \$	$T' \rightarrow *FT'$
\$ E' T' F *	* + i \$	Pop *
\$ E' T F	+ i \$	Synch , pop F
\$ E' T'	+ i \$	$T' \rightarrow \epsilon$
\$ E'	+ i \$	$E' \rightarrow +TE'$
\$ E' T +	+ i \$	Pop +
\$ E' T	i \$	$T \rightarrow FT'$

$\$ E' T' F$	$i \$$	$F \rightarrow i$
$\$ E' T' i$	$i \$$	$\text{Pop } i$
$\$ E' T'$	$\$$	$T' \rightarrow \epsilon$
$\$ E'$	$\$$	$E' \rightarrow \epsilon$
$\$$	$\$$	Recover