



Lectures in :

# Compilers Principles & Techniques

By

Dr. Esam T. Yassen

Uni. Al-Anbar, Computers College

Adapted By

Dr. Sameeh Abdulghafour Jasim

Uni. of AL-MAARIF, Col. of  
Sciences

## Introduction

### Programming languages :-

Interactions involving humans are most effectively carried out through the medium of language . language permits the expression of thoughts and ideas , and without it , communication as we know it would be very difficult indeed .

In computer programming , programming language serves as means of communication between the person with a problem and the computer used to solve it . programming language is a set of symbols , words , and rules used to instruct the computer .

A hierarchy of programming languages based on increasing machine independence include the following :-

**1- machine language :** is the actual language in which the computer carries out the instructions of program . otherwise , " it is the lowest form of computer language , each instruction in program is represented by numeric code , and numeric addresses are used throughout the program to refer to memory location in the computer memory .

**2- Assembly languages :** is a symbolic version of a machine language ,each operation code is given a symbolic code such as **Add** , **SUB** ,.... Moreover , memory location are given symbolic name , such as **PAY** , **RATE** .

**3-high – level language :**Is a programming language where the programmer does not require knowledge of the actual computing machine to write a program in the language .H.L.L . offer a more enriched set of language features such as control structures , nested statements , block ...

**4- problem-oriented language :** It provides for the expression of problems in a specific application . Examples of such language are **SQL** for Database application and **COGO** for civil engineering applications .

**Advantages of H.L.L over L.L.L include the following :**

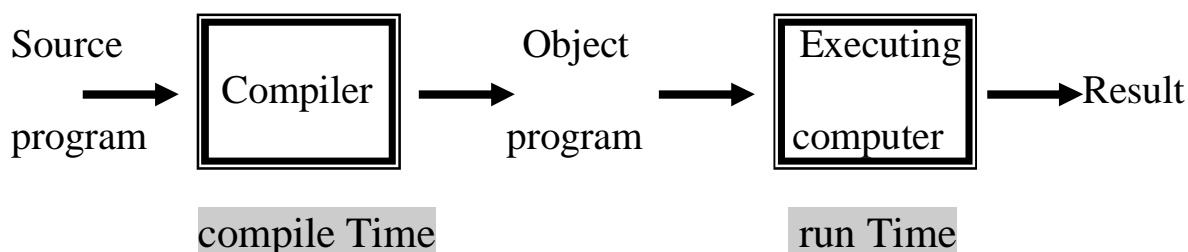
- 1- **H.L.L** are easier to learn then **L.L.L**
- 2- A programmer is not required to know how to convert data from external from to internal within memory .
- 3- Most **H.L.L** offer a programmer a variety of control structures which are not available in **L.L.L**
- 4- Programs written in **H.L.L** are usually more easily **debugged** than **L.L.L.** equivalents.
- 5- Most **H.L.L** offer more powerful data structure than **L.L.L.**
- 6- Finally ,High level languages are relatively **machine-independent**. Consequently certain programs are portable

**Translator:** High- Level language programs must be translated automatically to equivalent machine- language programs .

A translator input and then converts a " source program" into an object or target program . the source program is written in a source language and the object program belong to an object language .



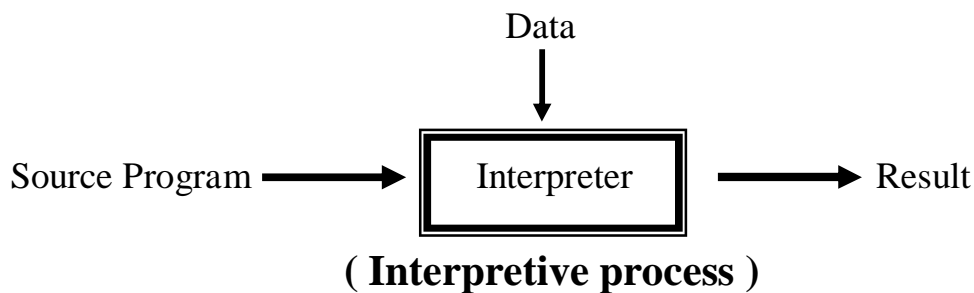
- 1- If the source program is written in *assembly language* and the target program in machine language .the translator is called "**Assembler** "
- 2- If the source language is **H.L.L.** and the object language is **L.L.L.** ,then the translator is called "**Compiler** " .
- 3- If the source language is **L.L.L.** and the object language is **H.L.L.**, then the translator is called "**Decompiler**"



**(compilation Process)**

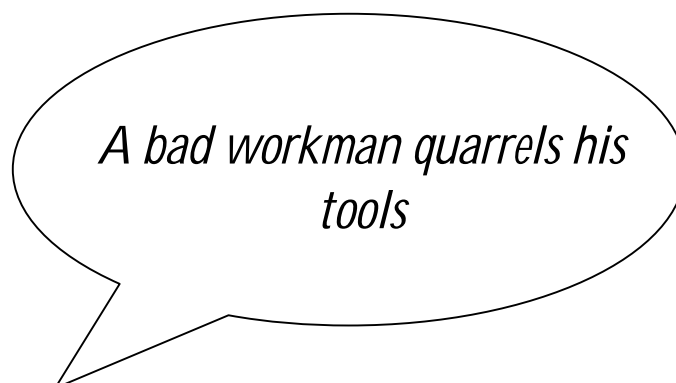
- The time at which conversion of a source program to an object program occurs is called " **Compile time** ". The object program is executed at " **Run time** ", note that the source program and data are processed at different times .

Another kind of translator ,called an " **Interpreter** " in which processes an internal form of source program and data at the same time . that is interpretation of the internal source from occurs at run time and no object program is generated .



Compiled programs usually run faster than interpreter ones because the overhead of understanding and translating has already been done .However ,Interpreters are frequent easier to write than Compilers , and can more easily support interactive debugging of program .

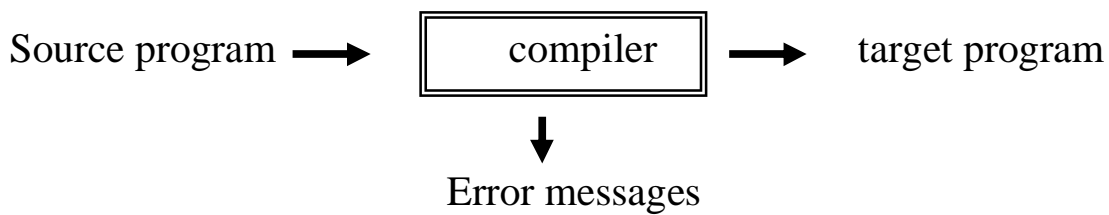
**Remark** :Some programming language implementations support both interpretation and compilation.



## **Compilation concepts**

### **What is compiler?**

A compiler is a program that translates a computer program(source program) written in H.L.L (such as Pascal,C++) into an equivalent program (target program) written in L.L.L.



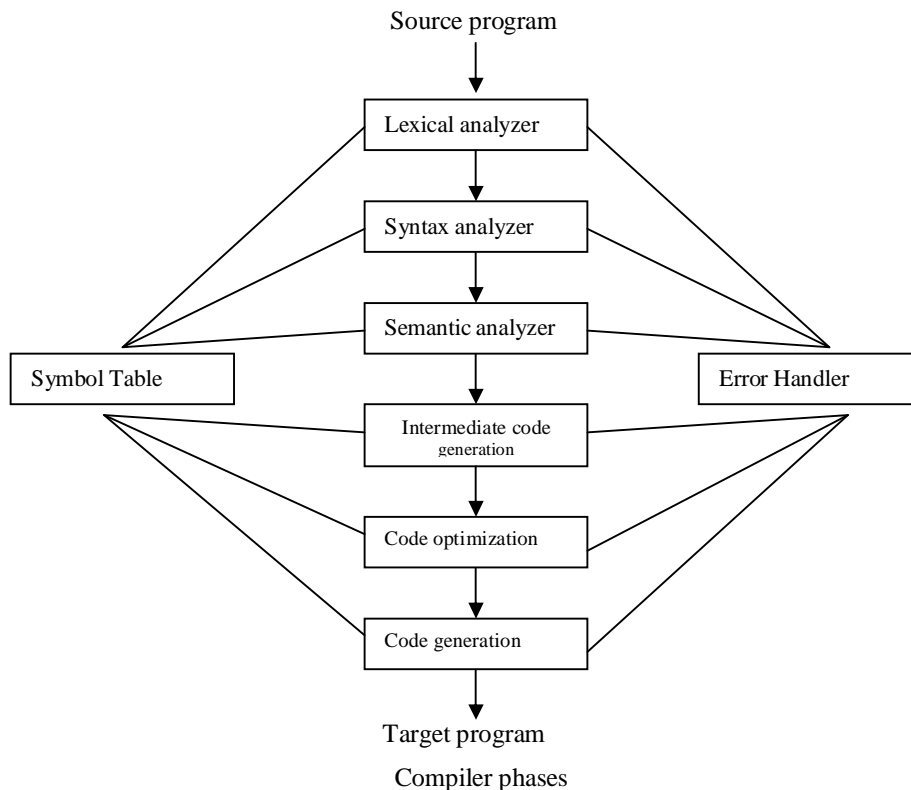
### **Model of Compiler:**

The task of constructing a compiler for a particular source language is complex. The complexity of the compilation process depend on the source language.A compiler must perform two major tasks:

1. Analysis :deals with the decomposition of the source program into its basic parts.
2. Synthesis:builds their equivalent object program using these basic parts.

To perform these tasks, compiler operates in phases each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in the following figure.

*Compilers*  
*Principle , Techniques, and Tools*



1. **Lexical Analyzer:** whose purpose is to separate the incoming source code into small pieces (tokens) , each representing a single atomic unit of language, for instance "keywords", "Constant ", " Variable name" and "Operators".
2. **Syntax Analyzer :** whose purpose is to combine the tokens into well formed expressions (statements) and program and it check the syntax error
3. **Semantic Analyzer:** whose function is to determine the meaning of the source program.
4. **Intermediate Code Generator:** at this point an internal form of a program is usually created. For example:

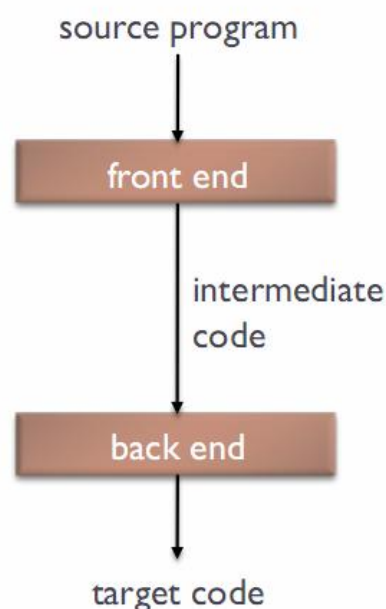
$Y=(a+b)*(c+b)$ 
➤
  
 (+,a,b,t1)
   
 (+,c,d,t2)
   
 (\*,t1,t2,t3)

*Compilers*  
*Principle ,Techniques, and Tools*

5. **Code Optimizer** :Its purpose is to produce a more efficient object program (Run faster **or** take less space **or** both)
6. **Code Generator**: Finally, the transformed intermediate representation is translated into the target language.

The grouping of phases : the phases of compiler are collection into :

1. **Front-End** :It consists of those phases that depend on the *source language* and are largely independent of the *target machine* ,those include : (**lexical analysis ,syntax analysis , semantic analysis, and intermediate code generation** )
2. **Back-End** : Includes those phases of compiler that depend on the *target machine* and not depend on the *source language* . these include:( **code optimization phase and code generation phase** )



**The grouping of Compiler Phases**

**Symbol–Table Management:** An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier .These attributes may provide information about the storage allocated for an identifier , its type and in case of procedure , the number and types of its arguments and so on .

Symbol-Table is a data structure containing a record for each identifier , with fields for the attributes of the identifier.

### **Error Detection and Reporting**

Each phase can encounter errors. However ,after defection an error a phase must somehow deal with that error, so the compilation can proceed, allowing further errors in the source program to be detected .A compiler that stops where it finds the first error is not as helpful as it could be.

The syntax and semantic analysis phases usually handle a large fraction of the error detectable by the compiler .

- **Types of Errors**

**Lexical errors:** The lexical phase can detect errors where the characters remaining in the input do not form any token of the language.

**Syntax errors:** The syntax analysis phase can detect errors Errors where the token stream violates the structure rules (syntax) of the language .

**Semantic errors:** During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved, e.g. to add two identifiers, one of which is the name of an array, and the other the name of a procedure .



*Compilers*  
*Principle ,Techniques, and Tools*

## Where errors show themselves

### Compile-time errors

Many errors are detected by the compiler, the compiler will generate an error message - Most compiler errors have a file name , line number, and Type of error. This tells you where the error was detected .

File name (fname.java)

```
C:\code javac fname.java
fname.java:23: cannot resolve symbol
symbol   : class string
location: class fname
    string msg;
    ^
1 error
```

Line number (23)

```
C:\code javac fname.java
fname.java:23: cannot resolve symbol
symbol   : class string
location: class fname
    string msg;
    ^
```

Type of error

```
C:\code javac fname.java
fname.java:23: cannot resolve symbol
symbol   : class string
location: class fname
    string msg;
    ^
1 error
```

*Compilers*  
*Principle ,Techniques, and Tools*

**Runtime Errors**

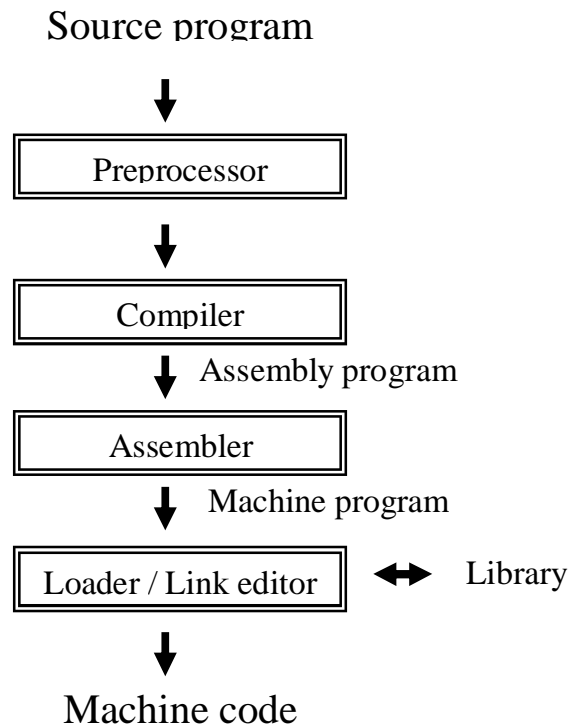
Runtime errors occur while the program is running, although the compilation is successful. The causes of Runtime Errors are [5]:

- 1) Errors that only become apparent during the course of execution of the program
- 2) External Factors – e.g.
  - Out of memory
  - Hard disk full
  - Insufficient i/o privileges
  - etc.
- 3) Internal Factors – e.g.
  - Arithmetic errors
  - Attempts to read beyond the end of a file
  - Attempt to open a non-existent file
  - Attempts to read beyond the end of an array
  - etc.

*Compilers*  
*Principle ,Techniques, and Tools*

### A Language- Processing System :-

In addition to a compiler ,several other programs may be required to create an executable target program.



### (Language- Processing System)

**Preprocessing** : During this stage , *comments* ,*macros* and *directives* are processed :

- *Comments* are removed from the source file.
- *Macros* :If the language supports macros ,the macros are replaced with the equivalent text,Example:

```
# define pi 3.14
```

When the preprocessor encounter the word(pi) it would replace (pi) with ( 3.14 )

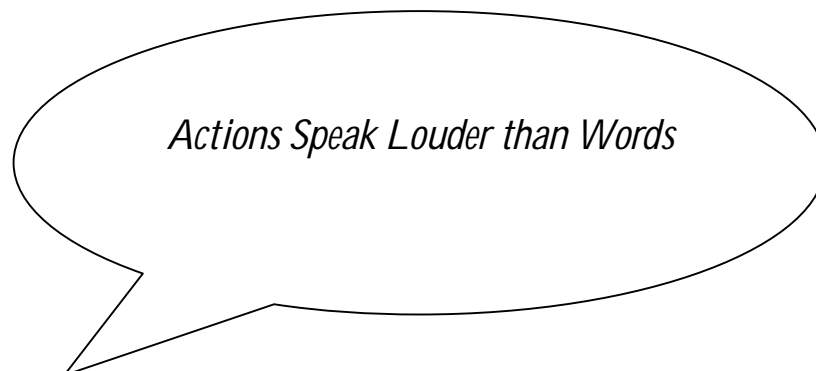
- *Directives* : The preprocessor also handles directives. In 'C' language , including statement looks like:

`# include<"file">`

this line is replaced by the actual file.

**Loader - Link Editor** : Is program that performs two functions:

1. **Loading** :taking relocatable machine code and placed the altered instructions and data in memory at the proper locations.
2. **Link-Editing** :Allows us to make a single program from several files . these files may have been result of different compilers and one or more may be library files.



## Lexical Analyzer

The analysis of source program during compilation is often complex . The construction of compiler can often be made easier if the analysis of source program is separated into two parts , with one part identifying the low – level language constructs , such as *variable names* , *keyword* , *labels* , and *operations* , and the second part determine the syntactic organization of the program .

**Lexical Analyzer** : the job of the lexical analyzer , or *scanner* , is to read the source program ,one character at a time and produce as output a stream of *tokens* . the tokens produced by the scanner serve as input the next phase , *parser* . Thus , the lexical analyzers job is the translate the source program into a form more conducive the recognition by the parser .

**Tokens** : are used to represent low – level program units such as:-

- *Identifiers* , such as *sum* , *value* , and *X* .
- *Numeric literals* , such as **123** and **1.35e02** .
- *Operators* , such as *+* , *\** , *&&* , *<=* , and *%* .
- *Keywords* , such as *if* , *else* and *returns*.
- Many other language symbols .

There are many ways we could represent the tokens of a programming language . one possibility is to use a 2- duple of the form **< token – class, value >** .

For example :-

- The identifiers *sum* and *value* may be represented as :  
**< ident , “ sum “ >**  
**< ident , “ value” >**
- The numeric literals *123* and *1.35E02* may be represented as :  
**< numericlital , “ 123” >**  
**< numericliteral , “ 1.35E02” >**
- The operators *>=* and *+* may be represented as :

< relop , “ >= “ >

< addop , “ + “ >

- The *scanner* may take the expression  $x = 2+f(3)$  , and produce the following stream of *tokens* :

< ident , “ x “ >

< assign – op , “ = “ >

< numlit , “ 2 “ >

< addop , “ + “ >

< ident , “ f ” >

< lparent , “ ( “ >

< numlit , “ 3 “ >

< rparent , “ ) “ >

< semicolon , “ ; “ >

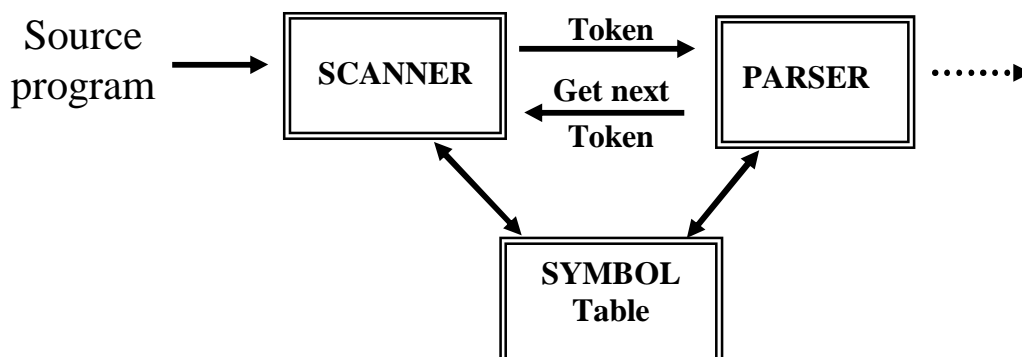
### Interaction of Scanner with Parser :

Using only *parser* can become costly in terms of **time** and **memory requirements** .The complexity and time can be reduced by using a *scanner* .

The separation of *scanner* and *parser* can have other advantages, scanning characters is typically slow in compilers and separating it from parsing particular emphasis can be given to making the process efficient .

Therefore, The *scanner* usually interacts with the *parser* in one of two ways :-

- 1- The *scanner* may process the source program in separate pass before parsing begins . Thus the *tokens* are stored in **file or large table** .
- 2- The second way involves an **interaction** between the *parser* and *scanner* , the *scanner* called by the *parser* whenever the next *token* in the source program is required .



**Interaction of Scanner with Parser**

The latter approach is the preferred method of operation , since an internal form of the complete source program dose not need to be constructed and stored in memory before parsing can begin .

**Note :** The lexical analyzer may also perform certain secondary tasks at the user interface : such task is stripping out from source program comments and white space in the form of bank , tab and new line characters.

**Lexical Errors** : the lexical phase can detect errors where the characters remaining in the input do not form any token of the language for example if the string “ fi “ is encountered in ‘ C ‘ program :-

fi ( A = = f(x) ) ...

A lexical analyzer can not tell whether “ fi “ is misspelling of the keyword “ if “ or an undeclared function identifier since “ fi “ is a valid identifier , the lexical must return the token for an identifier and let some other phase of compiler handle any error. The possible error – recovery actions are :

1. Deleting an extraneous character .
2. Inserting a missing character .
3. Replacing an incorrect character by a correct char .
4. Transposing two adjacent characters .

Finally , the scanner breaks the source program into tokens . the type of token is usually represented in the form of unique internal representation number or constant. For example, a variable name may be represented by 1 ,a constant by 2 , a label by 3 and so on .

The scanner then returns the internal type of token and some time the location in the table where the tokens are stored . Not all tokens may be associated with location , while variable name and constant are stored in table , operators , for example , may not be .

**Example :** Suppose that the value of tokens are :

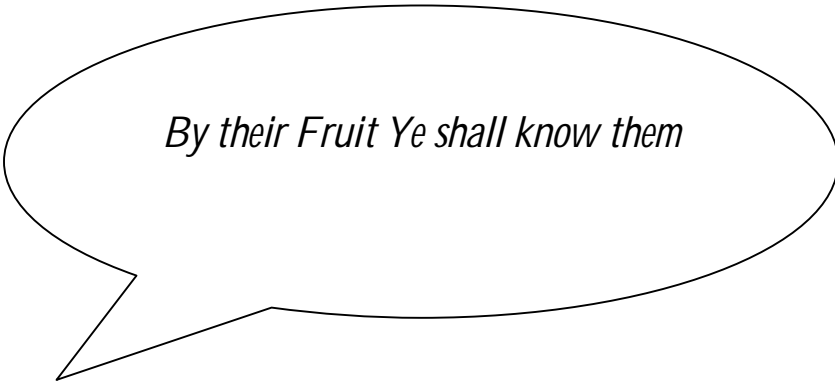
Variable name \_\_\_\_ 1  
Constant \_\_\_\_\_ 2  
Label \_\_\_\_\_ 3  
Keyword \_\_\_\_\_ 4  
Add operator \_\_\_\_\_ 5  
Assignment \_\_\_\_ 6

and the program is :

Sum : A = a+b ;  
Goto Done ;

**The output is :**

<u>Token</u>	<u>Internal represent</u>	<u>Location</u>
Sum	3	1
:	11	0
A	1	2
=	6	0
A	1	2
+	5	0
B	1	3
;	12	0
Goto	4	0
Done	3	4
;	12	0



*By their Fruit Ye shall know them*



## Symbol Table ( ST )

A *symbol table* is a data structure containing a record for each identifier, with fields for attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

When an identifier in source program is detected by the *lexical analyzer*, the identifier is entered into *ST*. However, the attributes of an identifier can not be determined during lexical analysis, the remaining phases enter information about identifier into *ST* and then use this information in various ways.

### Symbol Table Contents :-

A symbol table is most often conceptualized as series of rows, each row containing a list of attributes values that are associated with a particular variable. The kinds of attributes appearing in *ST* are dependent to some degree on the nature of language for which compiler is written. For example, a language may be typeless, and therefore the type attribute need not appear in *ST*. The following list of attribute are not necessary for all compilers, however, each should be considered for a particular compiler:-

- 1- **Variable Name:** A variable's name most always reside in the *ST*. major problem in *ST* organization can be the variability in the length of identifier names. For languages such as BASIC with its one – and two – character names and FORTRAN with names up to six characters in length, this problem is minimal and can usually be handled by storing the complete identifier in a fixed – size maximum length fields. While there are many ways of handling the storage of variable names, two popular approaches will be outlined, one which facilitates quick table access and another which supports the efficient storage of variable names. The provide quick access, yet sufficiently large, maximum variable name length. A length of sixteen or greater is very likely adequate (ملائم), the complete identifier can then be stored in a fixed – length fields in

*ST*, in this approach, table access is fast but the storage of short variable names is inefficient.

A second approach is to place a string **Descriptor** in the name field of the table. The descriptor contains (*position and length*) subfields. The pointer subfield indicates the position of the first character of the name in a general string area, and the length subfield describes the number of characters in the name. therefore, this approach results in slow table access, but the savings in storage can be considerable.

**2- Object Time Address:-** The relative location for values of variable at run time.

**3- Type:-** This field is stored in *ST* when compiling language having either *implicit* or *explicit* data type. For *typeless* language such as "BASIC" this attribute is excluded. "FORTRAN" provides an example of what mean by *implicit* data typing. Variables which are not declared to be particular type are assigned default types implicitly (variables with names starting with I, J, K, L, M, or N are *integer*, all other variables are *real*).

**4- Dimension of array or Number of parameters for a procedure.**

**5- Source line number at which the variable is declared.**

**6- Source line number at which the variable is referenced.**

**7- Link field for listing in alphabetical order.**

### **Operation on *ST*:-**

The two operations that are most commonly performed on *ST* are: ***Insertion & Lookup (Retrieval)*** .For language in which explicit declaration of all variables is mandatory (إجباري), an insertion is required when processing a declaration. If *ST* is *Ordered*, then insertion may also involve a lookup operation to find allocations at which the variable's attributes are to be placed. In such a situation an insertion is

at least as expensive as retrieval. If the *ST* is not ordered, the insertion is simplified but the retrieval is expensive.

Retrieval operations are performed for all references to variables which don't involve declaration statements.

The retrieved information is used for *semantic checking* and *code generation*. Retrieval operations for variables which have not been previously declared are detected at this stage and appropriate error messages can be emitted. Some recovery from such semantic errors can be achieved by posting a warning message and incorporation (ينشئ) the nondeclared variable in *ST*.

When a programming language permits implicit declarations of variable reference must be treated as an initial reference, since there is no way of knowing a priori of the variable's attributes have been entered in *ST*. Hence any variable reference generates a lookup operation followed by an insertion if the variable's name is not found in *ST*.

For *block – structured languages*, two additional operations are required: **Set & Reset**.

The Set operation is invoked when the beginning of a block is recognized during compilation. The complementary operation, the Reset operation is applied when the end of block is encountered. Upon block entry, the set operation establishes a new subtable (within the *ST*) in which the attributes for the variables declared in the new block can be stored. Because an new subtable is established for each block, the duplicated variable- name problem can be resolve.

Upon block exit the reset operation removes the subtable entries for the variables of the completed block.

### **ST Organizations:-**

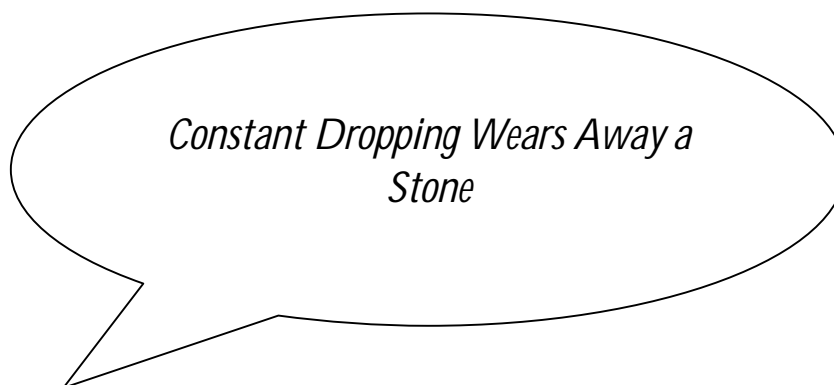
The primary measure which is used to determine the complexity of a *ST* operation is the average length of search. This measure is the average number of comparisons required to *Retrieve* a *ST* record in a particular table organization, the

name of variable for which an insertion or lookup operation is to be performed will be referred to as the search argument.

**1- Unordered ST:** The simplest method of organization *ST*, is to add the attribute entries to the table in the order in which the variable are declared. In an insertion operation no comparisons are required.

**2- Ordered ST :** In this and following organization, we described *ST* organization in which the table position of a variable's set of attributes is based on the variable's name. An *insertion* operation must be accompanied by *lookup* procedure which determines where in *ST* the variables attribute should be placed. The insertion of new of attributes may generate some additional overhead primarily because other sets of attributes may have to be moved in order to a chive the insertion.

**3- Tree – structured ST :**The time to performed an insertion operation can be reduced by using a tree – structured type of storage organization.

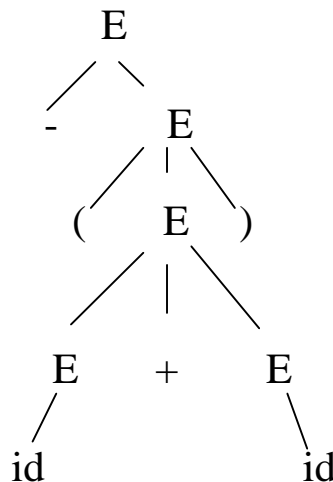


### Syntactic Analyzer ( Parser )

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 grammars. In syntax analysis we are concerned with grouping *tokens* into larger syntactic classes such as *expression* , *statements* , and *procedure*. The syntax analyzer (parser) outputs a *syntax tree*, in which its leaves are the *tokens* and every non-leaf node represents a syntactic *class* type. For example:- Consider the following grammars:-

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

Then the parse tree for **-(id+id)** is:-



### Syntax Error Handling :-

Often much of the error detection and recovery in a compiler is central around the *parser*. One reason of this is that many errors are syntactic in nature. Errors where the token stream violates the structure of the language are determined by parser, such as an arithmetic expression with unbalanced parentheses.

**Derivations :-**

This derivational of view gives a precise description of the *top-down* construction of *parse tree*. The central idea here is that a *production* is treated as rewriting rule in which the *nonterminal* on the left is replaced by the string on the right side of the *production*. For example, consider the following grammar:

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

The *derivation* of the input string **id + id\* id** is:

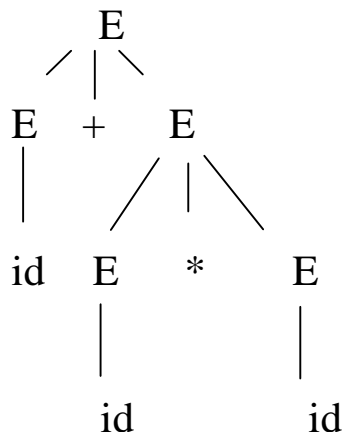
**Left-most derivation**

E  
E+E  
id +E  
id+E\*E  
id+id\*E  
id+id\*id

**Right-most derivation**

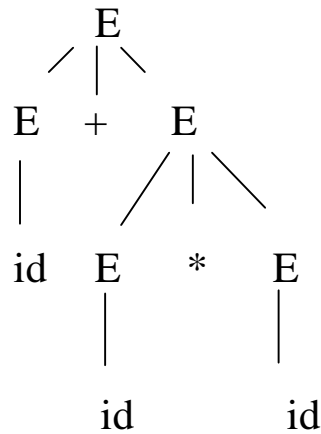
E  
E+E  
E+E\*E  
E+E\*id  
E+id\*id  
id+id\*id

Note:- *parse tree* may be viewed as a graphical representation for a derivation :

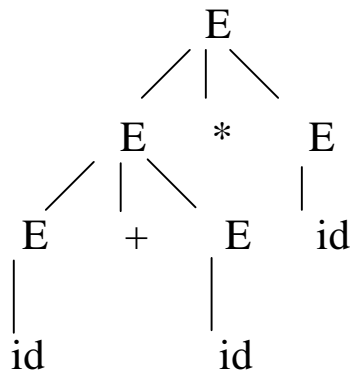


**Ambiguity :-**

A grammar that produce more that one parse tree for same sentence is said to be **Ambiguous**. In the another way, by produced more that one *left-most derivation* or more that one *Right-most derivation* for the same sentence.



(1)



(2)

two parse tree for **id+id\*id**

E  
E+E  
id+E  
id+E\*E  
id\*id\*E  
id+id\*id

(1)



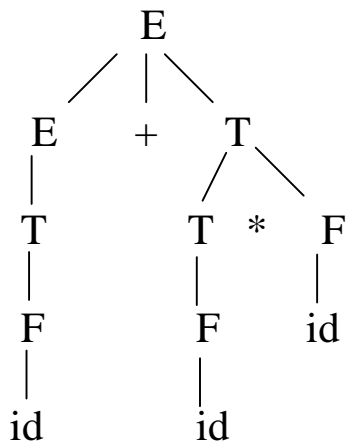
E  
E\*E  
E+E\*E  
id+E\*E  
id+id\*E  
id+id\*id

(2)

Two *left-most derivations* for **id+id\*id**  
With My Best Wishes

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

$E \rightarrow E+E$	}	$E \rightarrow E+T \mid T$
$E \rightarrow E * E$		$E \rightarrow (E)$
$E \rightarrow (E)$		$E \rightarrow -E$
$E \rightarrow -E$		$T \rightarrow T * F \mid F$
$E \rightarrow id$		$F \rightarrow id$



parse tree for **id+id\*id**

**Left-Recursion :-**

A grammar is left-recursion if has a *nonterminal* **A** such that there is a derivation  $A \rightarrow A\alpha$  for some string  $\alpha$ . Top-down parser cannot handle *left-recursion* grammars, so a transformation that eliminates left-recursion is needed:


$A \rightarrow A\alpha \mid \mathbf{B}$	}
$A \rightarrow \mathbf{B} A'$	
$A' \rightarrow \alpha A' \mid \epsilon$	

**OR:**



$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \dots | \beta_1 | \beta_2 | \dots | \beta_n$$

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

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon$$


**Example:**

$$E \rightarrow E+T | T$$

$$T \rightarrow T*F | F$$

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

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' | \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *F T' | \epsilon$$

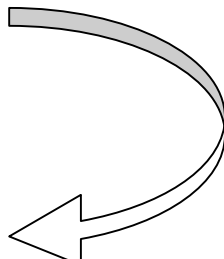
$$F \rightarrow (E) | id$$

**Left-Factoring :-**

The basic idea is that when is not clear which of two alternative production 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.

$$A \rightarrow \alpha \beta_1 | \alpha \beta_2 \quad \text{where } \alpha \neq \epsilon$$

$$A \rightarrow \alpha A'$$

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


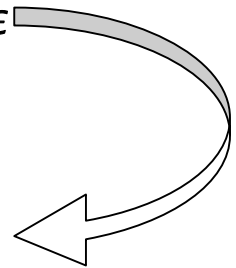
OR :-

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \gamma$$

where  $\alpha \neq \epsilon$

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

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



Example:

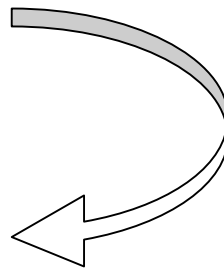
$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

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

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

$$E \rightarrow b$$



*Easy Come, Easy Go*

### Top-Down Parsing :-

Top-down parsing can be viewed as an attempt to find a *leftmost derivation* for an input string. Equivalently, A top down parser, such as LL(1) parsing, move from the goal symbol to a string of terminal symbols. in the terminology of trees, this is moving from the root of the tree to a set of the leaves in the syntax tree for a program. in using full backup we are willing to attempt to create a syntax tree by following branches until the correct set of terminals is reached. in the worst possible case, that of trying to parse a string which is not in the language, all possible combinations are attempted before the failure to parse is recognized. the nature of top down parsing technique is characterized by:

**1-Recursive-Descent Parsing :** It is a general form of Top-Down Parsing that may involve " *Backtracking* ",that is ,making repeated scans of the input.

**Example:** consider the grammar

$$S \longrightarrow cAd$$

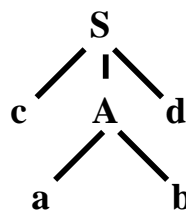
$$A \longrightarrow ab \mid a$$

Input : **cad**

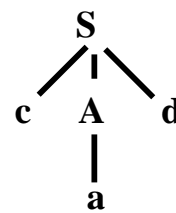
Then the implementation of Recursive-Descent Parsing is:



- a -



- b -



- c -

**2-Predictive parsing :** In many cases, by carefully writing a grammar , eliminating *left-recursion* from it and *left-factoring* the resulting grammar, we can obtain a grammar that can be parsed by *recursive-descent parser* that needs no "*Backtracking*",i.e.,a **Predictive parser**.

## 2.1. Transition Diagrams for Predictive parsers

It is useful plan or flowchart for a predictive parser. There is one diagram for each *nonterminal*, the labels of edges are *tokens* and *nonterminals*.for example:

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

$$T \rightarrow T*F \mid F \quad // \text{Original grammar}$$

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

**Eliminate left-recursion and left factoring**

$$E \rightarrow T E'$$

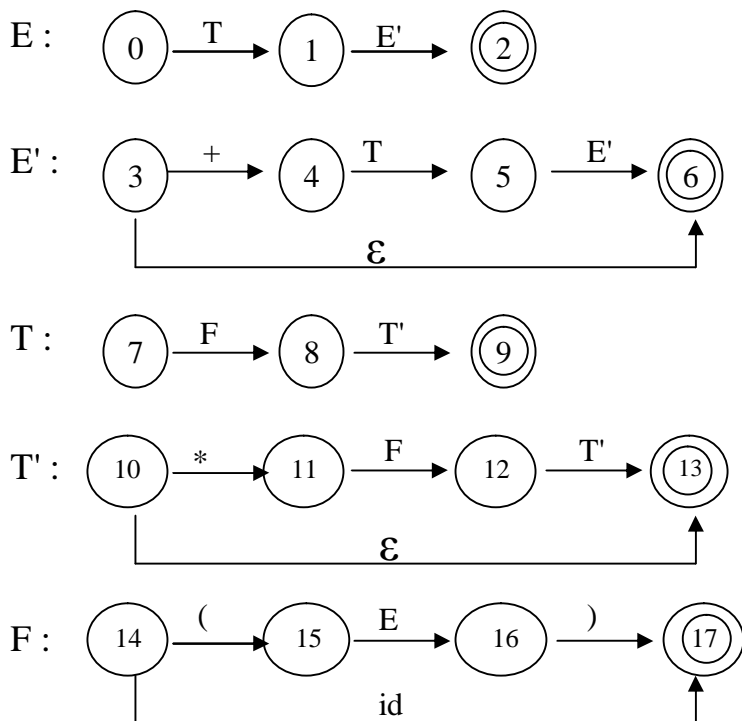
$$E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow FT'$$

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

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

### Transition Diagrams



**First & Follow :**

- **First :** To compute First(X) for all grammar symbols apply the following rules until no more *terminal* or  $\epsilon$  can be added to any First set :
  1. If x is *terminal*, then **FIRST(x)** is {x}.
  2. If  $x \rightarrow \epsilon$  is a production ,then add  $\epsilon$  to FIRST(x).
  3. If x is *nonterminal* and  $x \rightarrow y_1y_2 \dots y_k$  is a production, then place *a* in FIRST(x) if for some *i* ,*a* is in FIRST(*y<sub>i</sub>*),and  $\epsilon$  is in all of FIRST(*y<sub>1</sub>*)... FIRST(*y<sub>i-1</sub>*).
- **Follow :**To compute Follow(A) for all *nonterminals* apply the following rules until nothing can be added to any Follow set.
  1. Place \$ in FOLLOW(S),where S is the start symbol.
  2. If there are a production  $A \rightarrow \alpha B \beta$ , then everything in FIRST( $\beta$ )except for  $\epsilon$  is placed in FOLLOW(B).
  3. If there are a production  $A \rightarrow \alpha B$  ,or a production  $A \rightarrow \alpha B \beta$  where FIRST( $\beta$ ) contains  $\epsilon$ , then everything in FOLLOW(A) is in FOLLOW(B).

Example : suppose the following grammar

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

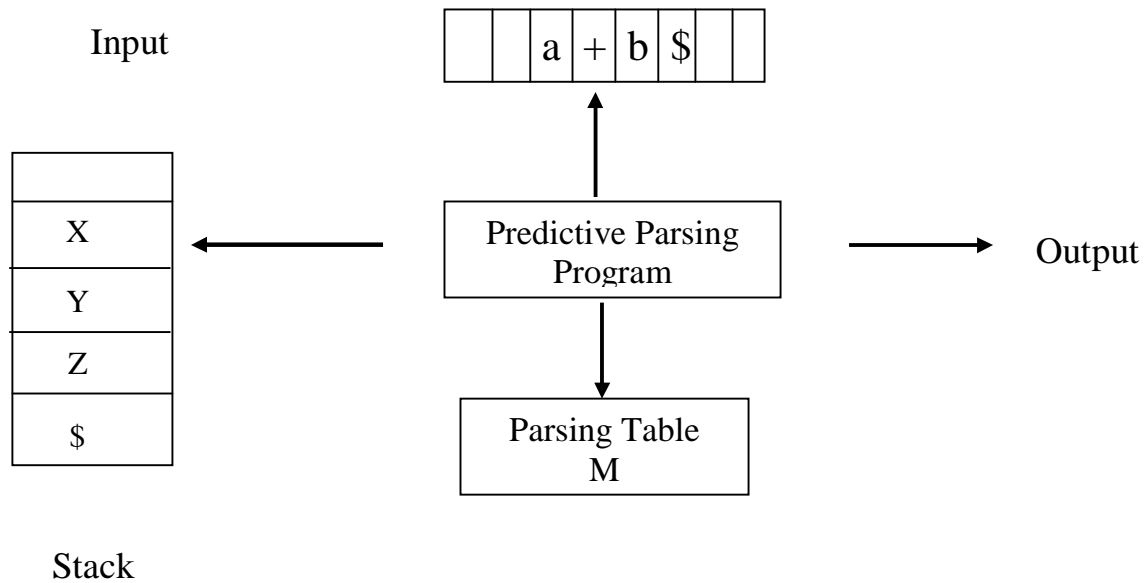
$$T \rightarrow F T'$$

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

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

Nonterminals	First	Follow
<b>E</b>	( , id	) , \$
<b>E'</b>	+ , $\epsilon$	) , \$
<b>T</b>	( , id	+ , ) , \$
<b>T'</b>	* , $\epsilon$	+ , ) , \$
<b>F</b>	( , id	* , + , ) , \$

**2.2. Nonrecursive Predictive Parsing :-**The nonrecursive parser in following figure lookup the production to be applied in a parsing table.



### Model of a Nonrecursive Predictive Parsing

- **Construction of Predictive Parsing Table :**

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

- **Predictive Parsing Program :**The parser is controlled by a program that behaves as follows:  
**The program consider X-** the symbol on top of the stack- and  $a$  – the current input symbol-. These two symbols determine the action of the parser. There are three possibilities :

1. If  $X = a = \$$  , the parser halt, and 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 nonterminal , the program consults entry  $M[X,a]$  of the parsing table.If  $M[X,a]=\{ X \rightarrow UVW \}$  the parser replaces X on top of stack by WVU ( with U on top ).

**Example:**

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

$$T \rightarrow T * F \mid F \quad // \text{Original grammar}$$

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

**Eliminate left-recursion and left factoring**

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow FT'$$

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

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

**Predictive Parsing Table M**

Nonterminals	Input symbol					
	id	+	*	(	)	\$
<b>E</b>	TE'			TE'		
<b>E'</b>		+TE'			$\epsilon$	$\epsilon$
<b>T</b>	FT'			FT'		
<b>T'</b>		$\epsilon$	*FT'		$\epsilon$	$\epsilon$
<b>F</b>	id			(E)		

### Implement Predictive Parsing Program

stack	Input	output
\$E	id+id*id\$	
\$E'T	id+id*id\$	E → TE'
\$E'T'F	id+id*id\$	T → FT'
\$E'T'id	id+id*id\$	F → id
\$E'T'	+id*id\$	
\$E'	+id*id\$	T' → ε
\$E'T+	+id*id\$	E' → +TE'
\$E'T	id*id\$	
\$E'T'F	id*id\$	T → FT'
\$E'T'id	id*id\$	F → id
\$E'T'	*id\$	
\$E'T'F*	*id\$	T' → *FT'
\$E'T'F	id\$	
\$E'T'id	id\$	F → id
\$E'T'	\$	T' → ε
\$E'	\$	E' → ε
\$	\$	Accept

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

**Example :- (H.W)**

$$S \longrightarrow iEtSS' \mid a$$

$$S' \longrightarrow eS \mid \epsilon$$

$$E \longrightarrow b$$

*Everything Comes to him who Waits*