



Lectures in :

# Compilers Principles & Techniques

By

M.Sc. Esam T. Yassen

Uni. Al-Anbar, Computers College

## 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 ...

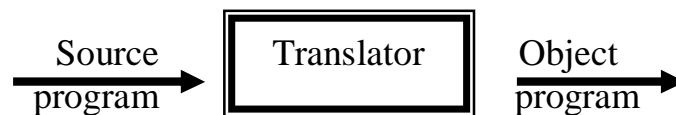
**4- problem-oriented language :** It provides for the expression of problems in a specific application . Examples of such language are **SQL** for Database application and **COGO** for civil engineering applications .

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

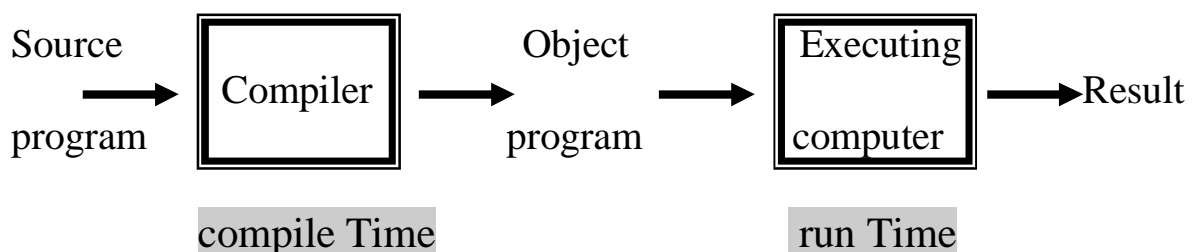
- 1- **H.L.L** are easier to learn then **L.L.L**
- 2- A programmer is not required to know how to convert data from external from to internal within memory .
- 3- Most **H.L.L** offer a programmer a variety of control structures which are not available in **L.L.L**
- 4- Programs written in **H.L.L** 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 .



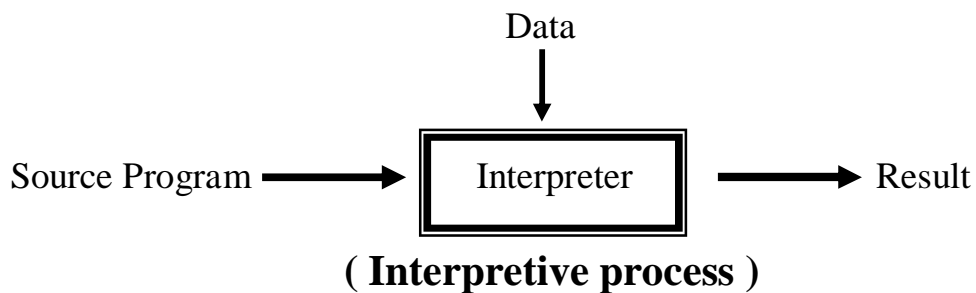
- 1- If the source program is written in *assembly language* and the target program in machine language .the translator is called "**Assembler** "
- 2- If the source language is **H.L.L.** and the object language is **L.L.L.** ,then the translator is called "**Compiler** " .
- 3- If the source language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



**(compilation Process)**

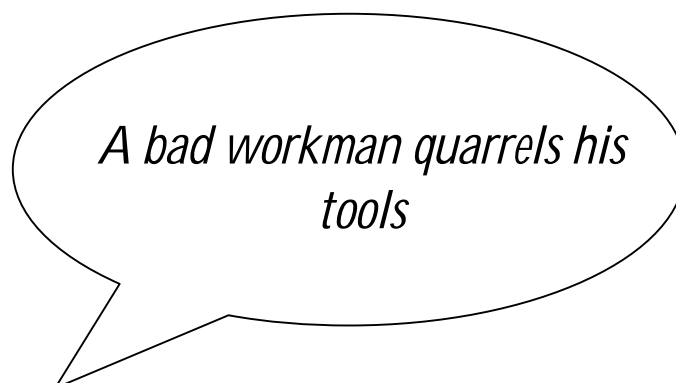
- The time at which conversion of a source program to an object program occurs is called " **Compile time** ". The object program is executed at " **Run time** ", note that the source program and data are processed at different times .

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 .

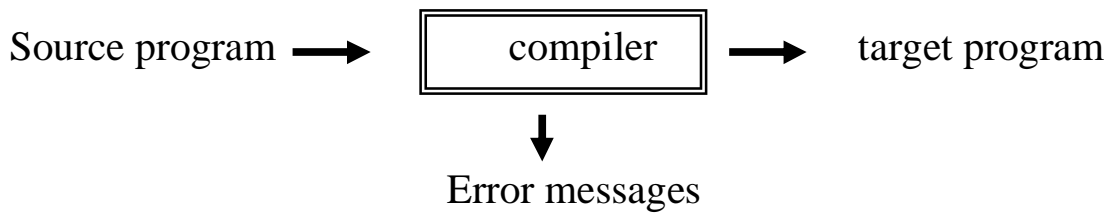
**Remark** :Some programming language implementations support both interpretation and compilation.



## **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.

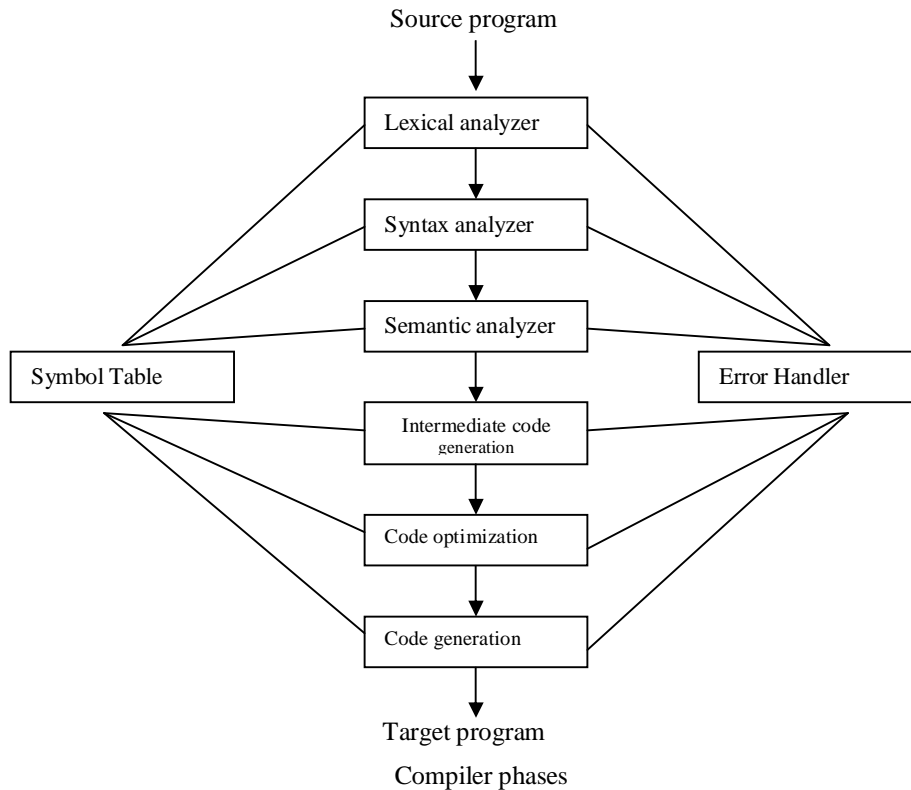


### **Model of Compiler:**

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.



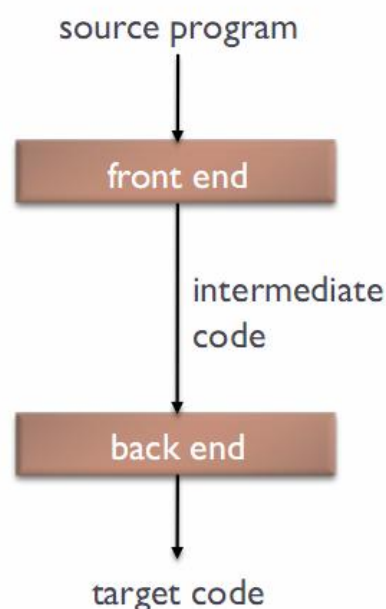
1. **Lexical Analyzer:** whose purpose is to separate the incoming source code into small pieces (tokens) , each representing a single atomic unit of language, for instance "keywords", "Constant ", " Variable name" and "Operators".
2. **Syntax Analyzer :** whose purpose is to combine the tokens into 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)

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 .



## 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
```

### **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

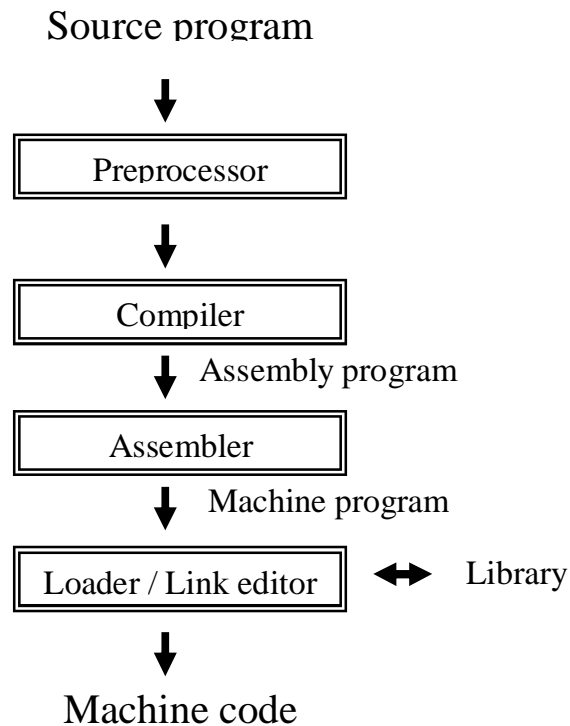
Attempt to open a non-existent file

Attempts to read beyond the end of an array

etc.

### 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 )

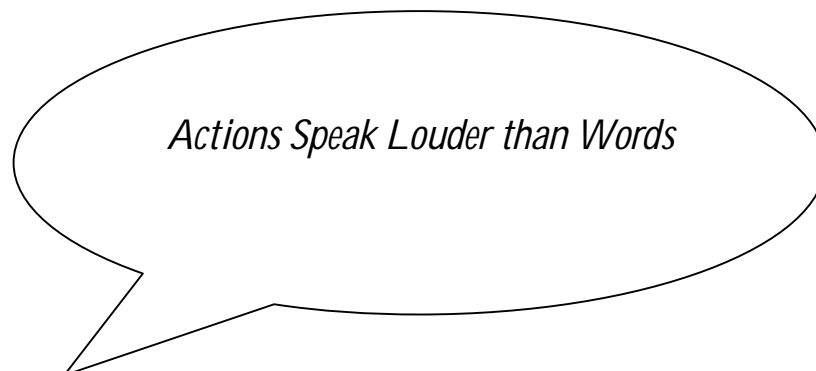
- *Directives* : The preprocessor also handles directives. In 'C' language , including statement looks like:

`# 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 “ >

< rporent , “ ) “ >

< semicolon , “ ; “ >

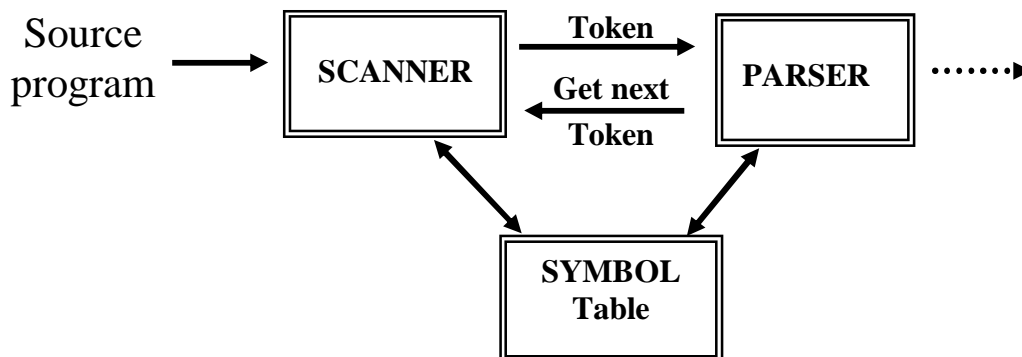
### Interaction of Scanner with Parser :

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 :

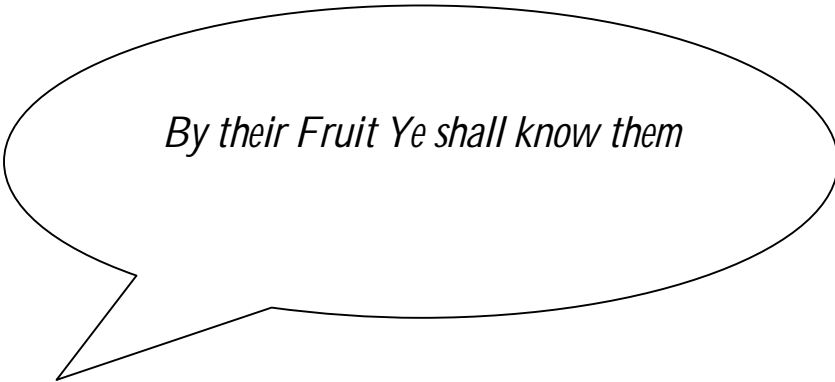
Variable name \_\_\_\_ 1  
Constant \_\_\_\_\_ 2  
Label \_\_\_\_\_ 3  
Keyword \_\_\_\_\_ 4  
Add operator \_\_\_\_\_ 5  
Assignment \_\_\_\_\_ 6

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.

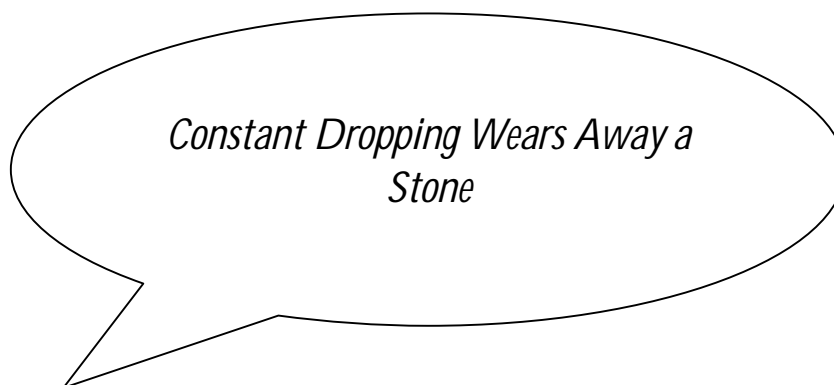
Upon block exit the reset operation removes the subtable entries for the variables of the completed block.

### **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.

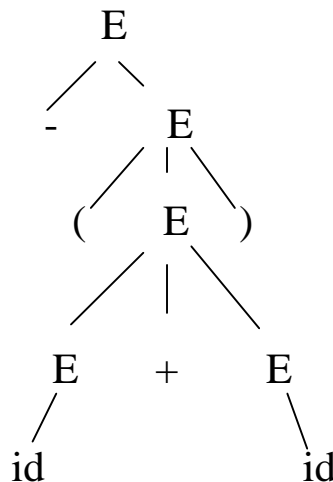


### Syntactic Analyzer ( Parser )

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:

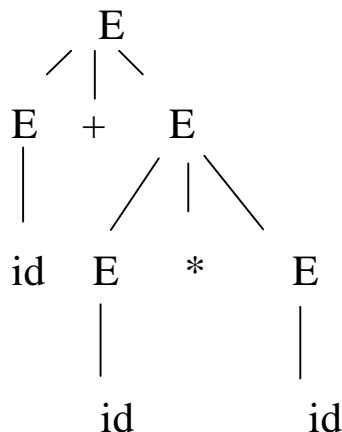
**Left-most derivation**

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

**Right-most derivation**

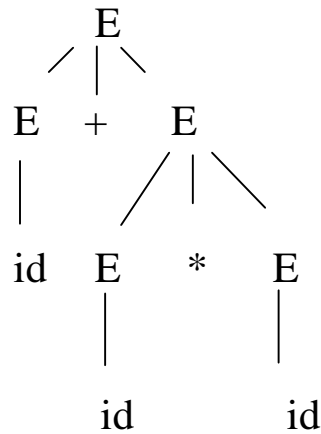
E  
E+E  
E+E\*E  
E+E\*id  
E+id\*id  
id+id\*id

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

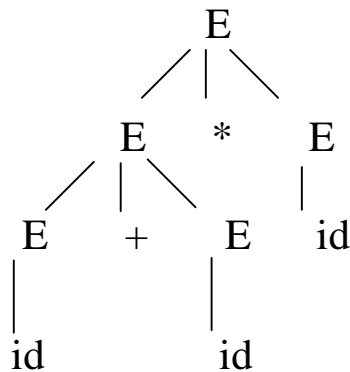


**Ambiguity :-**

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



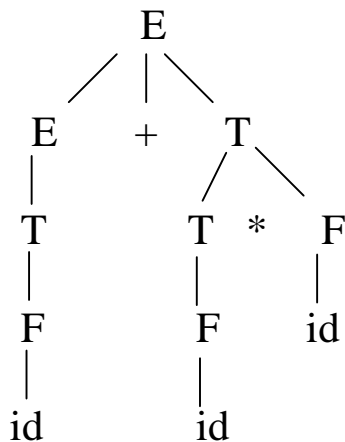
E  
E\*E  
E+E\*E  
id+E\*E  
id+id\*E  
id+id\*id

(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:**

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

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

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

$$E \rightarrow T E'$$

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

$$T \rightarrow FT'$$

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

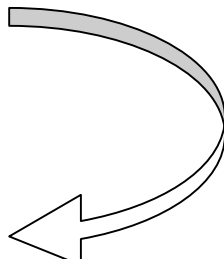
$$F \rightarrow (E) | id$$

**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.

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

$$A \rightarrow \alpha A'$$

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


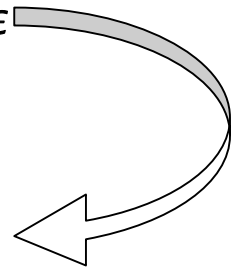
**OR :-**

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

where  $\alpha \neq \epsilon$

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

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



**Example:**

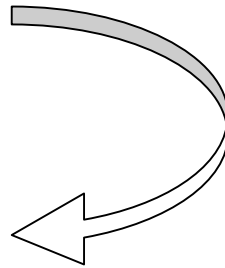
$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

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

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

$$E \rightarrow b$$



*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

$$S \longrightarrow cAd$$

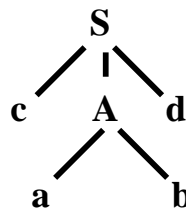
$$A \longrightarrow ab \mid a$$

Input : **cad**

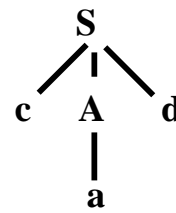
Then the implementation of Recursive-Descent Parsing is:



- 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**

$$E \rightarrow T E'$$

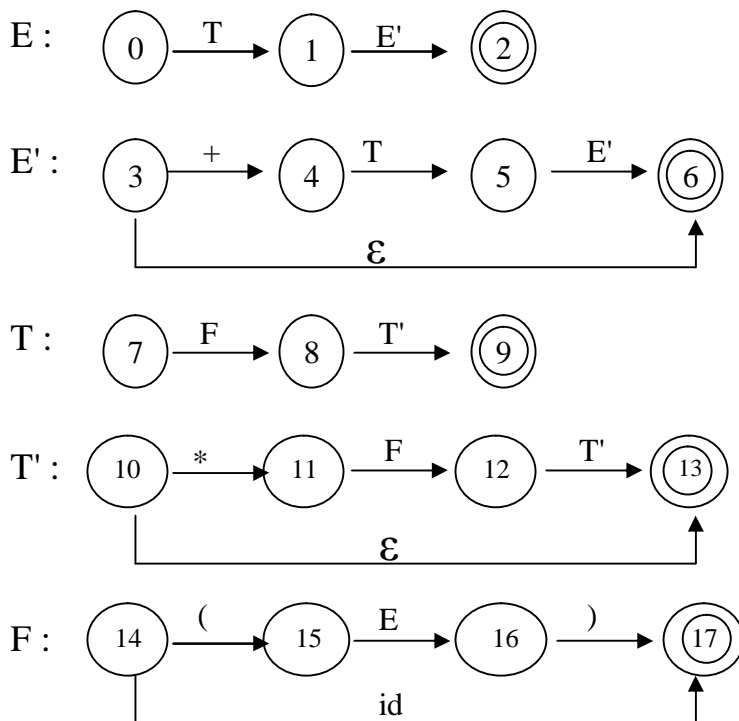
$$E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow FT'$$

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

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

### Transition Diagrams



### First & Follow :

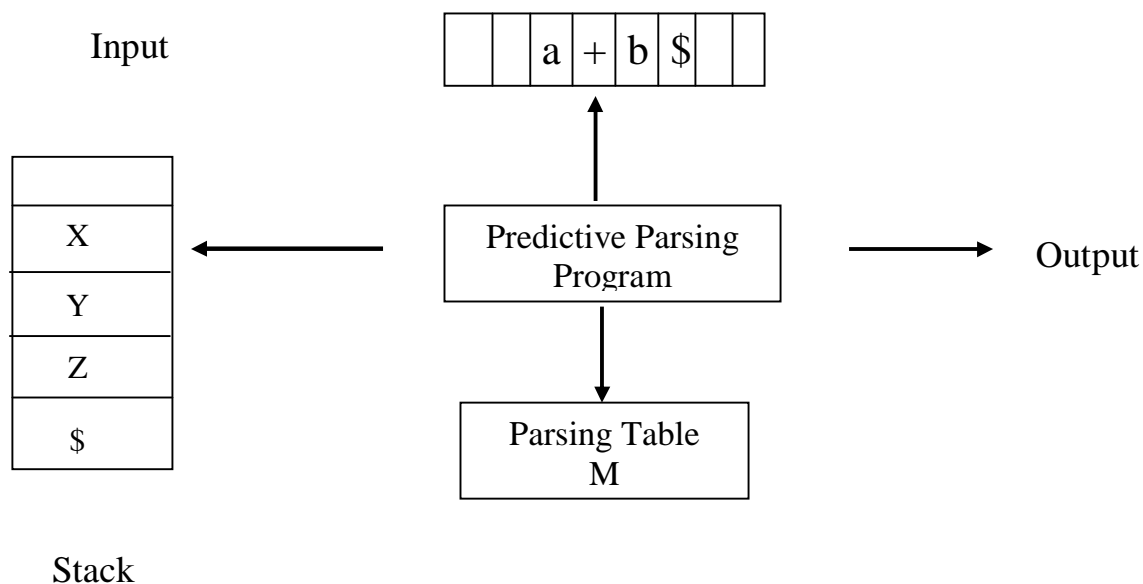
- **First :** To compute First(X) for all grammar symbols apply the following rules until no more *terminal* or  $\epsilon$  can be added to any First set :
  1. If x is *terminal*, then **FIRST(x)** is {x}.
  2. If  $x \rightarrow \epsilon$  is a production ,then add  $\epsilon$  to FIRST(x).
  3. If 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

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

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:**

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

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

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

**Eliminate left-recursion and left factoring**

$$E \rightarrow T E'$$

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

$$T \rightarrow FT'$$

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

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

**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*



## Bottom-Up Parsing

The term "Bottom-Up Parsing" refer to the order in which nodes in the parse tree are constructed, construction starts at the leaves and proceeds towards the root. Bottom-Up Parsing can handle a large class of grammars.

**1. Shift-Reduce Parsing:** Is a general style of Bottom-up syntax analysis , it attempts to construct a parse tree for an input string beginning at leaves and working up towards the root,(reducing a string  $w$  to the start symbol of grammar).At each reduction step a particular substring matching the right side of production is replaced by the symbol on the left of that production.

**Example :** consider the grammar

$$\begin{aligned} S &\longrightarrow aABe \\ A &\longrightarrow Abc \mid b \\ B &\longrightarrow d \end{aligned}$$

And the input is **abbcd e**

The implementation Bottom-Up Parsing is

a b b c d e  
a A b c d e  
a A d e  
a A B e  
S  
Accept

**Handle :** Is a substring that matches the right side of a production.

### Stack Implementation of Shift-Reduce Parsing:

A convenient way to implement a shift-reduce parser is to use a *Stack* to hold a grammar symbols and an input buffer to hold the sting  $w$  to be parsed. We use \$ to mark the bottom of *stack* and also the right end of the input string. There are actually four possible actions:

1. **Shift** : The next input symbol is Shifted onto the top of *stack*.
2. **Reduce** : Replace the handle with nonterminal.
3. **Accept** : The parser announces successful completion of parsing .
4. **Error** : The parser discovers that syntax error has occurred and calls an error recovery routine.

**Example:** Consider the following grammar

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

And the input string is **id + id \* id**, then the implementation is :

Stack	Input Buffer	Action
\$	id+id*id\$	Shift
\$id	+id*id\$	Reduce: E→id
\$E	+id*id\$	Shift
\$E+	id*id\$	Shift
\$E+id	*id\$	Reduce: E→id
\$E+E	*id\$	Shift(*)
\$E+E*	id\$	Shift
\$E+E*id	\$	Reduce: E→id
\$E+E*E	\$	Reduce: E→E*E
\$E+E	\$	Reduce: E→E+E
\$E	\$	Accept

### Conflicts During Shift-Reduce Parsing:

There are context free grammars for which shift-reduce parsing cannot be used. Ambiguous grammars lead to parsing conflicts. Can fix by rewriting grammar or by making appropriate choice of action during parsing. There are two type of conflicts :

1. **Shift/Reduce** conflicts: should we shift or reduce? (See previous example (\*))
2. **Reduce/Reduce** conflicts: which production should we reduce with? for example:

stmt  $\rightarrow$  id(param)  
param  $\rightarrow$  id  
expr  $\rightarrow$  id(expr) | id

<u>Stack</u>	<u>Input Buffer</u>	<u>Action</u>
\$...id(id	,id)...\$	Reduce by ??

Should we reduce to **param** or to **expr** ?



*Example is Better than Precept*

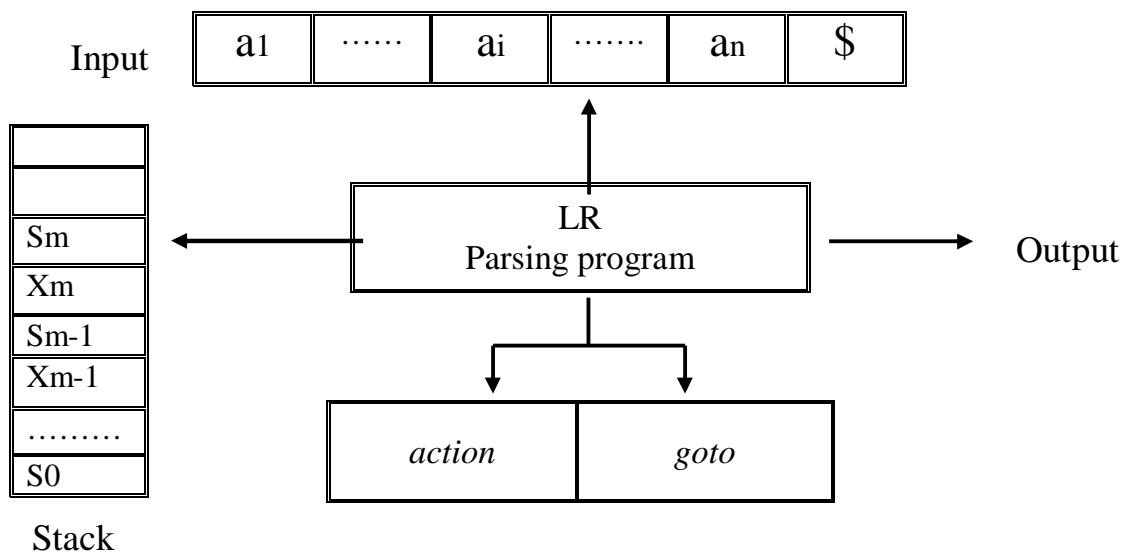
## LR Parsers

This section presents an efficient Bottom-Up syntax analysis technique that can be used to parse a large class of context-free grammars. The technique is called LR(k) parsing, the "L" is for left to right scanning of input, the "R" for constructing a rightmost derivation in reverse, and "k" for the number of input symbols of lookahead that are used in making parsing decisions-when "k" is omitted , k is assumed to be 1).

LR parsing is attractive for a variety of reasons:-

1. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
2. The LR parsing method is the most general *nonbacktracking* shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
3. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

The schematic form of an LR parser is shown in following figure .It consists of **an Input,an Output,a Stack,a Driver program,and a Parsing table** that has two parts (*action* and *goto*) .



**Model of an LR parser**

There are three techniques for LR parser depending on the construct of LR parsing table for a grammar :

1. **Simple LR parser (SLR for short):**Is the easiest to implement but the least powerful of the three.It may be fail to produce a parsing table for certain grammars on which the other methods succeed.
2. **Canonical LR parser:** It is most powerful, and most expensive.
3. **Lookahead LR parser (LALR for short):**It is intermediate in power and cost between other two. The LALR method will work on most programming-language grammars and ,with some effort ,can be implemented efficiently.

**The LR parsing Algorithm :-** The LR program is the same for all LR parsers, only the parsing table changes from one parser to another.

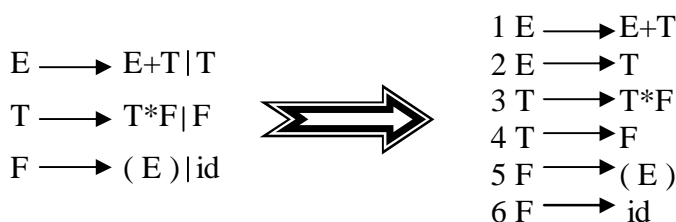
```
push the start state  $s_0$  onto the stack.
while (true) begin
  s = state on top of the stack and
  a = input symbol pointed to by input pointer ip
  if action[s,a] = shift s' then begin
    push a then s' onto the stack
    advance ip to the next input symbol
  end
  else if action[s,a] = reduce  $A \rightarrow \beta$  then begin
    pop  $2*|\beta|$  symbols off the stack, exposing state s'
    push A then goto[s',A] onto the stack
    output production  $A \rightarrow \beta$ 
  end
  else if action = accept then return
  else error()
end
```

### Implementation of SLR parser:-

The SLR-parser Extremely tedious to build by hand, so need a generator. The following steps represents the main stages, which are used to build system that is used for implementing the SLR-parser:

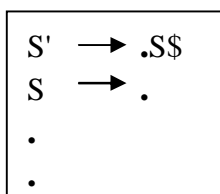
**1. Input stage :** In this state the grammar has been reading and the symbols of grammar (*terminals* and *nonterminals*) could be specified and each production of grammar must be on one straight line. Finally, the productions has been numbered.

For example, consider the grammar



**2. Compute First & Follow stage :** Through this state First & Follow could be detected for each *nonterminal*.

**3. Construct DFA stage:**By using a deterministic finite automaton ( DFA )the SLR-parser know when to *shift* and when to *reduce*. the edges of DFA are labeled by symbols of grammar( terminals & nonterminals).In this state, where the input begins with S'(root),that means that it begins with any possible right-hand side of an S-production we indicate that by



Call this **state1** or **state0**,a productions combined with the **dot(.)** that indicates a position of parser.Firstly ,for each production in state1 we exam the symbol that occur after dot, there are three cases :

1. If the symbol is **null** (the dot has been occurred in the end of right side of production),then there are no new state .
2. If the symbol is "\$" sign, then there are no new state.

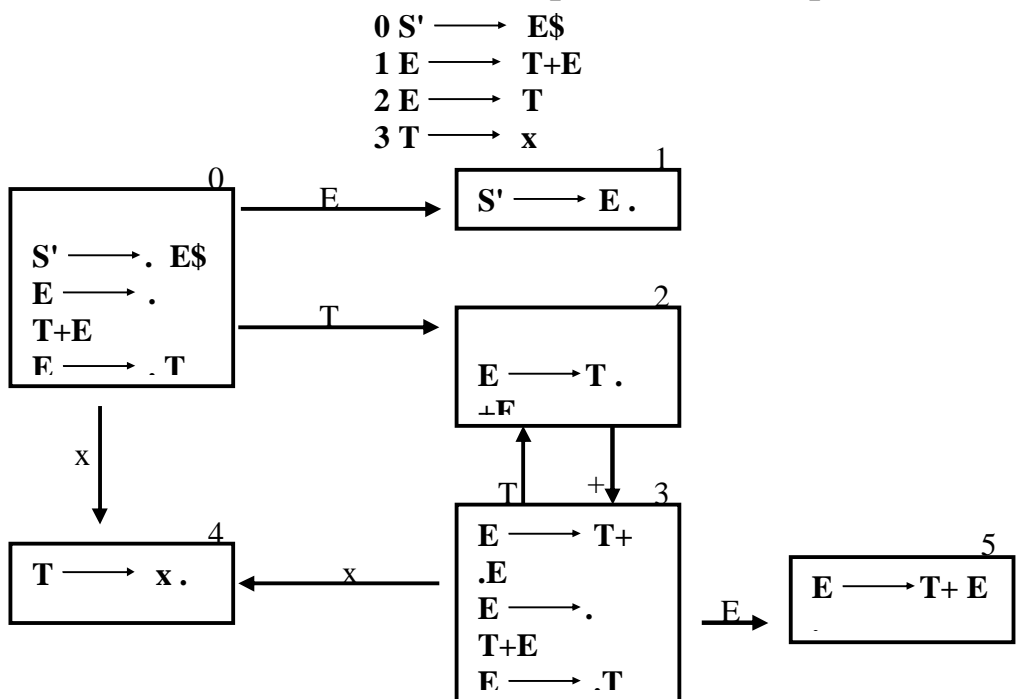
3. If the symbol is a *terminal* or *nonterminal*, then there are new state, this state start with current production after the dot has been proceeded one step forward. If the symbol has been occurred after the dot(in new position)is *nonterminal* such as *A*, then we add all possible right hand side of *A* to a new state, and so on.

You must know that any new state must built firstly in a buffer, and we compare it with a previous states in **DFA**, if there are no similarity situation then the new state is added to **DFA** and give it a new number equal to number of states in DFA plus one. Finally ,we repeat this steps on all new states until the **DFA** completed.

**Example** : consider grammar

$$\begin{aligned} E &\longrightarrow T+E \\ E &\longrightarrow T \\ T &\longrightarrow x \end{aligned}$$

Initially, it will have an empty stack, and the input will be a complete S-sentence followed by \$;that is the right-hand side of the S' rule will be on the input. we indicate this as  $S' \longrightarrow . S \$$  where the dot indicates the current position of the parser. So:



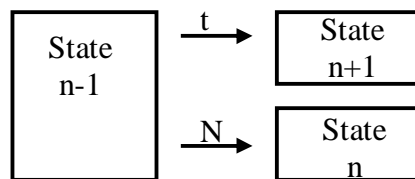
**4. Construct SLR table stage:** The SLR-table is a data structure consist of many rows equal to the number of the states in DFA, also many columns equal to the number of grammar symbols plus "\$" sign .As know ,data structure presents fast in information treatment and information retrieve .In this stage SLR-table is constructed .this table had seen as two subtables:

1. **The Action table:** consist of many rows equal to number of states in DFA, and many columns equal to number of terminals plus "\$" sign (the end of input).
2. **The Goto table:** consist of many rows equal to number of states in DFA, and many columns equal to number of nonterminals.

the elements (entries) in the SLR-table are labeled with four kinds of actions:

- $S_n$  shift into state  $n$
- $g_n$  goto state  $n$
- $r_k$  reduce by production  $k$
- $a$  accept
- error (denoted by blank entry in the table)

For the construction of this table and the contribution the actions on the tables cells must pass to each state in DFA individually :



- Shift action & Goto action could be specified according to the edge which has been moved from the current state ( $n$ ) to the new state.

If the edge was terminal symbol ( $t$ ) then

$$\text{Cell}[n-1, t] = s_n$$

If the edge was nonterminal symbol ( $N$ ) then

$$\text{Cell}[n-1, N] = g_n$$



- If there are production in current state has the form  $A \rightarrow \mathbf{B}$ . (the dot in the end of right hand side,  $\mathbf{B}$  is any string ),then the action is reduce  

$$\text{Cell}[n-1,f]=rk \quad \{ f \text{ in Follow}(A), k \text{ is the no. of production} \}$$
- If there are production in current state has the form  $A \rightarrow \mathbf{B}.$  {the dot occurred before \$ sign,  $\mathbf{B}$  is any string },then the action is accept  

$$\text{Cell}[n-1, \$]=a$$
- Finally, any empty cell in row n-1 means error action.  
Repeat the above steps for each states in DFA.

state	x	+	\$	E	T
0	S4			g1	g2
1			Accept		
2		S3	r2		
3	S4			g5	g2
4		r3	r3		
5			r1		

**5.Implement LR Algorithm :** Suppose input string is  $x+x$ . After insert input string the **LR-program** is executed, as follows:

Stack	Input	Action
0	$x+x \$$	shift
0S4	$+x\$$	Reduce by $T \rightarrow x$
0T2	$+x\$$	shift
0T2S3	$x\$$	Shift
0T2S3S4	$\$$	Reduce by $T \rightarrow x$
0T2S3T2	$\$$	Reduce by $E \rightarrow T$
0T2S3E5	$\$$	Reduce by $E \rightarrow T+E$
0E1	$\$$	Accept

## Semantic Analysis

The semantic analysis phase of compiler connects variable definition to their uses ,and checks that each expression has a correct type.

This checking called "**static type checking**" to distinguish it from "**dynamic type checking**" during execution of target program. This phase is characterized be the maintenance of symbol tables mapping identifiers to their types and locations.

### Examples of static type checking:-

- 1. Type checks :** A compiler should report an error if an operator is applied to an incompatible operand.
- 2. Flow of control checks:-** Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. For example, a "*break*" statement in 'C' language causes control to leave the smallest enclosing *while* , *for* , or *switch* statement ;an error occurs if such an enclosing statement does not exist.
- 3. Uniqueness checks:-** There are situations in which an object must be defined exactly once. For example, in 'Pascal' language, an identifier must be declared uniquely.
- 4. Name-related checks:-** Sometimes, the same name must appear two or more times. For example, in '**Ada**' language a loop or block may have a name that appear at the beginning and end of the construct. The compiler must check that the same name is used at both places.

### Type system:-

The design of type checker for a language is based on information about the syntactic constructs in the language, the notation of types, and the rules for assigning types to language constructs.

The following excerpts are examples of information that a compiler writer might have to start with.

- If both operands of the arithmetic operators "*addition*", "*subtraction*", and "*multiplication*" are of type *integer* , then the result is of type *integer*.

- The result of Unary & operator is a pointer to the object referred to by the operand. If the type of operand is  $T$  , the type of result is ' pointer to  $T$  '.

**We can classify type into :**

- 1. Basic type:** This type are the atomic types with no internal structure , such as *Boolean, Integer, Real, Char, Subrange, Enumerated*, and a special basic types " *type-error, void* ".
- 2. Construct types:** Many programming languages allows a programmer to construct types from *basic types* and other *constructed types*. For example *array, struct, set*.
- 3. Complex type:** Such as *link list, tree, pointer*.

**Type system:-** is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a *type system*.

**Specification of a simple type checker:-**

The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. In this section, we specify a type checker for simple language in which the type of each identifier must be declared before the identifier is used.

Suppose the following grammar to generates program, represented by *nonterminal* P, consisting of a sequence of declarations D followed by a single expression E.

$$P \longrightarrow D ; E$$

$$D \longrightarrow D ; D \mid id : T$$

$$T \longrightarrow char \mid int \mid array[num] of T \mid \uparrow T$$

$$E \longrightarrow literal \mid num \mid id \mid E mod E \mid E[E] \mid E \uparrow$$

Type checker ( translation scheme) produce the following part that saves the type of an identifier:

$P \longrightarrow$	$D;E$	
$D \longrightarrow$	$D;D$	
$D \longrightarrow$	$id:T$	$\{ \text{addtype}(id.entry, T.type) \}$
$T \longrightarrow$	$char$	$\{ T.type=char \}$
$T \longrightarrow$	$int$	$\{ T.type=int \}$
$T \longrightarrow$	$\uparrow T1$	$\{ T.type=pointer(T1.type) \}$
$T \longrightarrow$	$array[num] \text{ of } T1$	$\{ T.type=array(1..num.val, T1.type) \}$

- **The type checking of expression:** the following some of semantic rules:

$E \longrightarrow$	$literal$	$\{ E.type=char \}$	//constants represented
$E \longrightarrow$	$num$	$\{ E.type=int \}$	// = =

We can use a function *lookup( e )* to fetch the type saved in *ST* ,if identifier " e " appears in an expression:

$E \longrightarrow$	$id$	$\{ E.type=lookup(id.entry) \}$
---------------------	------	---------------------------------

The following expression formed by applying (mod) to two subexpression:

$E \longrightarrow$	$E1 \text{ mod } E2$	$\{ E.type= \text{if } E1.type=int \text{ and } E2.type=int \text{ then } int$ $\text{Else type-error } \}$
---------------------	----------------------	--

An array reference:

$E \longrightarrow$	$E1[E2]$	$\{ E.type= \text{if } E2.type=int \text{ and } E1.type=array[s,t] \text{ then } t$ $\text{Else type-error } \}$
---------------------	----------	---

$E \longrightarrow$	$E1 \uparrow$	$\{ E.type= \text{if } E1.type=pointer(t) \text{ then } t$ $\text{Else type-error } \}$
---------------------	---------------	--

- **The type checking of statements :**

$S \longrightarrow$	$id=E$	$\{ S.type= \text{if } id.type = E.type \text{ then } void$ $\text{Else type-error } \}$
---------------------	--------	---

$S \longrightarrow$	$\text{if } E \text{ then } S1$	$\{ S.type= \text{if } E.type=boolean \text{ then } S1.type$ $\text{Else type-error } \}$
---------------------	---------------------------------	--

$S \longrightarrow$	$\text{while } E \text{ do } S1$	$\{ S.type= \text{if } E.type=boolean \text{ then } S1.type$ $\text{Else type-error } \}$
---------------------	----------------------------------	--

$S \longrightarrow$	$S1 ; S2$	$\{ S.type= \text{if } S1.type=void \text{ and } S2.type=void \text{ then } void$ $\text{Else type-error } \}$
---------------------	-----------	---

## Intermediate Code Generation ( IR)

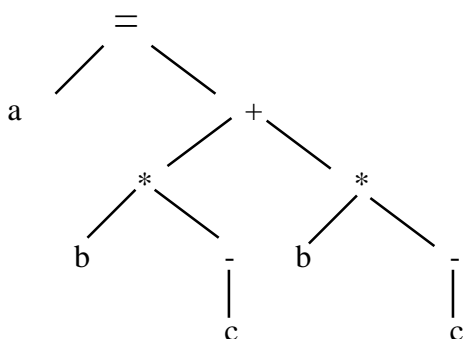
IR is an internal form of a program created by the compiler while translating the program from a *H.L.L* to *L.L.L* (*assembly* or *machine code*), from IR the back end of compiler generates *target code*.

Although a source program can be translated directly into the target language, some benefits of using a machine independent IR are:

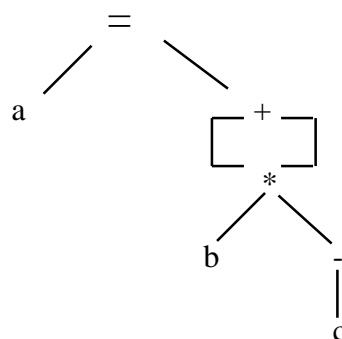
1. A compiler for different machine can be created by attaching a back end for a new machine into an existing front end.
2. Certain optimization strategies can be more easily performed on IR than on either original program or L.L.L.
3. An IR represents a more attractive form of target code.

### Intermediate Languages:-

1. Syntax Tree and Postfix Notation are two kinds of intermediate representations, for example  $a = b * -c + b * -c$

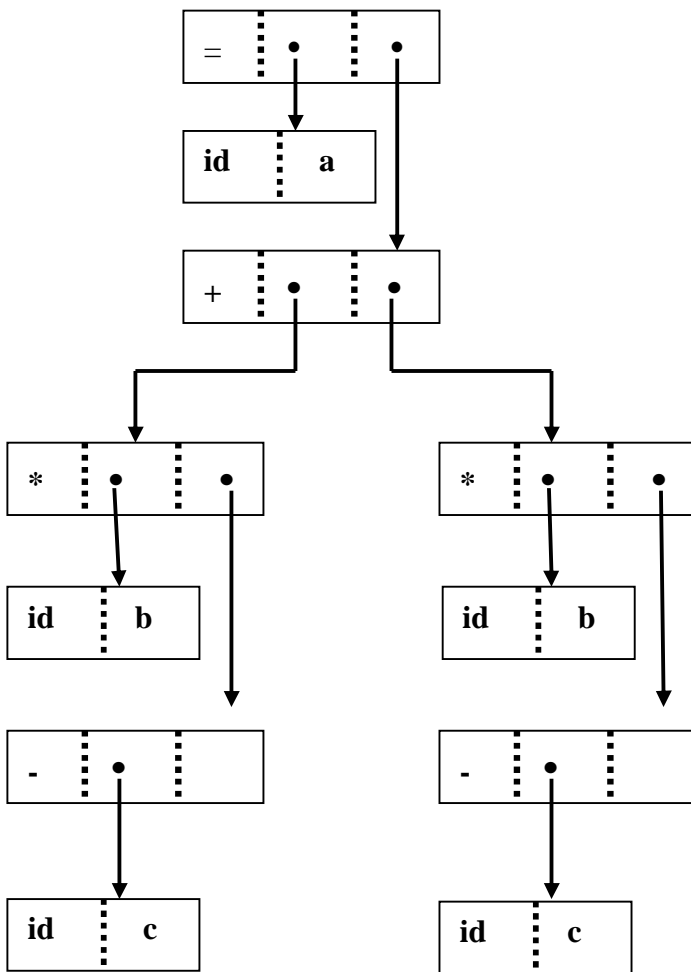


**Syntax Tree**



**DAG**

- A *DAG* give the same information in syntax tree but in compact way because common subexpressions are identified.
- *Postfix notation* is a linearized representation of a syntax tree, for example:  $a \ b \ c \ - \ * \ b \ c \ - \ * \ + \ =$
- Two representation of above syntax tree are:



1

0	id	b	
1	id	c	
2	-	1	
3	*	0	2
4	id	b	
5	id	c	
6	-	5	
7	*	4	6
8	+	3	7
9	id	a	
10	=	9	8
	....	....	....
	....	....	....

2

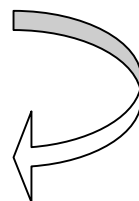
2. Three-Address Code is a sequence of statements of the general form :

$$X = Y \text{ op } Z \quad // \text{ op is binary arithmetic operation}$$

For example :  $x + y * z$

$$t1 = y * z$$

$$t2 = x + t1$$



where t1 ,t2 are compiler generated temporary.

**Types of three address code statement:-**

1. Assignment statements of the form  $X=Y \text{ op } Z$  ( where  $\text{op}$  is a binary arithmetic or logical operator).
2. Assignment instructions of the form  $X= \text{op } Y$  (  $\text{op}$  is a unary operator).
3. Copy statements of the form  $X=Y$  .
4. Unconditional jump ( *Goto L* ).
5. Conditional jump ( *if X relop Y goto L* ).
6. *Param X & Call P,N* for procedure call and and return  $Y$  , for example :

Param x1  
Param x2  
.....  
Param xn  
Call P,n

7. Index assignments of the form  $X=Y[i]$  &  $X[i]=Y$ .
8. Address & Pointer Assignments

$X= \&Y$   
 $X= * Y$   
 $*X= Y$

Example :  $a= b * -c + b * -c$

**t1 = - c**  
**t2 = b \* t1**  
**t3 = - c**  
**t4 = b \* t3**  
**t5 = t2 + t4**  
**a = t5**

**Three address code**  
**For syntax tree**

**t1 = - c**  
**t2 = b \* t1**  
**t5 = t2 + t2**  
**a = t5**

**Three address code**  
**For DAG**

Note: Three-address statements are a kin to assembly code statements can have symbolic labels and there are statements for flow of control.

### **Implementation of Three Address Code :-**

In compiler , three-address code can be implement as records, with fields for operator and operands.

**1. Quadruples :-** It is a record structure with four fields:

- **OP** // operator
- **arg1 , arg2** // operands
- **result**

**2. Triples :-** To avoid entering temporary into *ST* , we might refer to a temporary value by position of the statement that compute it . So three address can be represent by record with only three fields:

- **OP** // operator
- **arg1 , arg2** // operands



**Example:  $a = b * -c + b * -c$**

**i. By Quadruples**

Position	OP	arg1	arg2	result
0	-	c		t1
1	*	b	t1	t2
2	-	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

**ii. By Triples**

Position	OP	arg1	arg2
0	-	c	
1	*	b	(0)
2	-	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

## Code Optimization

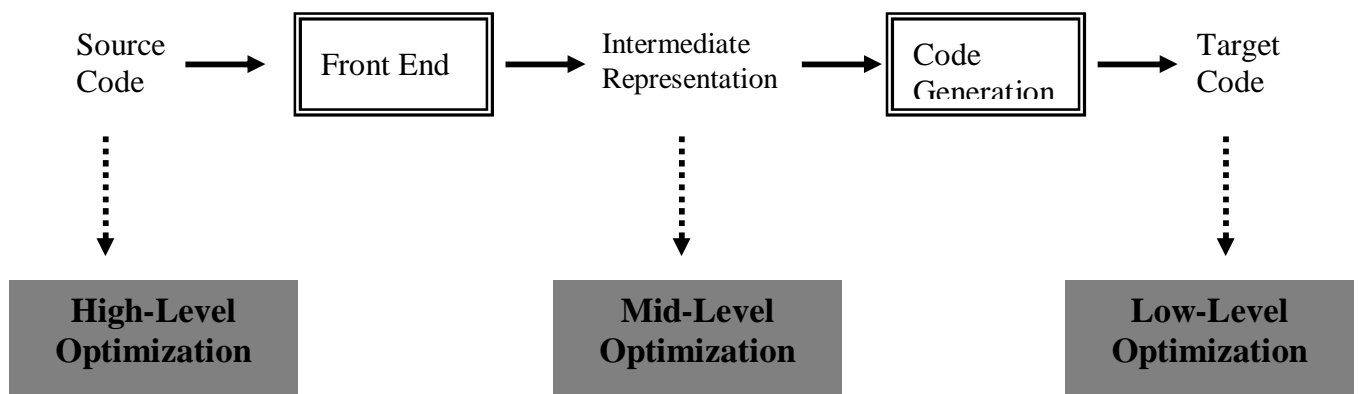
Compilers should produce target code that is as good as can be written by hand. This goal is achieved by program transformations that are called " Optimization ". Compilers that apply code improving transformations are called " Optimizing Compilers " .

Code optimization attempts to increase program efficiency by restructuring code to simplify instruction sequences and take advantage of machine specific features:-

- Run Faster , or
- Less Space , or
- Both ( Run Faster & Less Space ) .

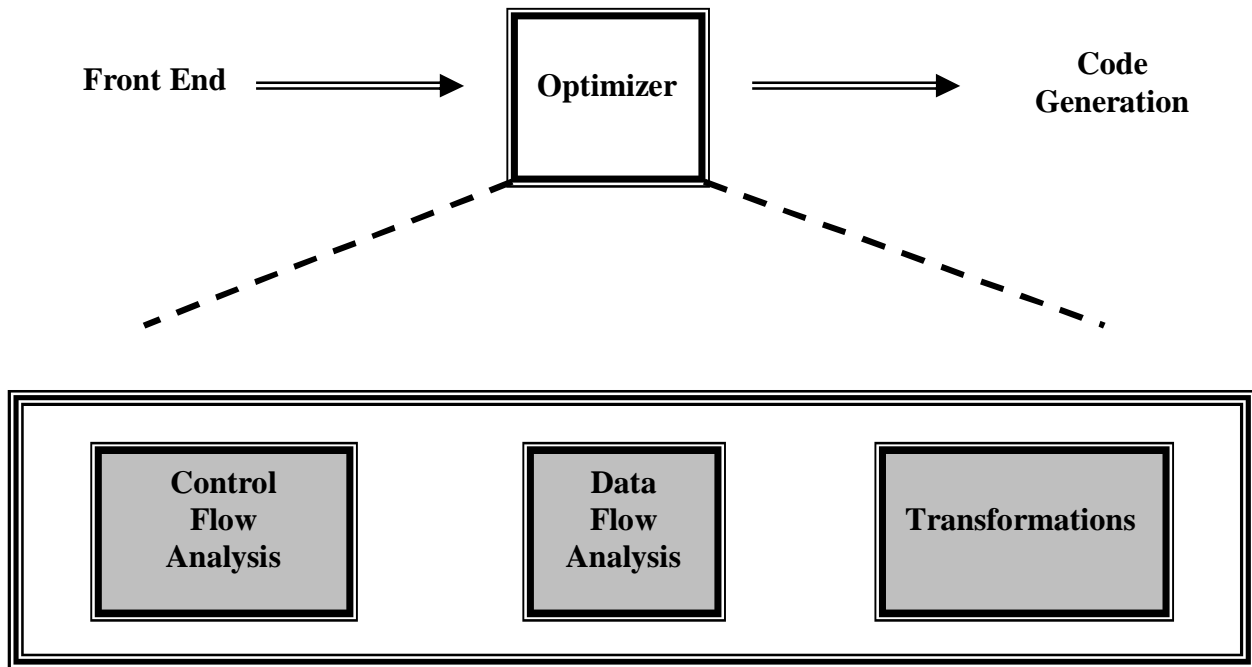
The transformations that are provided by an optimizing compiler should have several properties:-

1. A transformation must preserve the meaning of program. That is , an optimizer must not change the output produce by program for an given input, such as **division by zero**.
2. A transformation must speed up programs by a measurable amount.



## Places for Optimization

This lecture concentrates on the transformation of intermediate code ( Mid-Optimization or Independent Optimization ), this optimization using the following organization:-



### Organization of the Optimizer

This organization has the following advantages :-

1. The operations needed to implement high-level constructs are made explicit in the intermediate code.
2. The intermediate code can be independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for different machine.

### Basic Blocks:-

The code is typically divided into a sequence of "Basic Blocks". A Basic Block is a sequence of straight-line code, with no branches "In" or "Out" except a branch "In" at the top of block and a branch "Out" at the bottom of block.

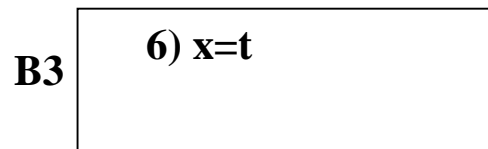
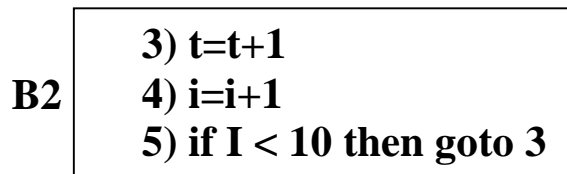
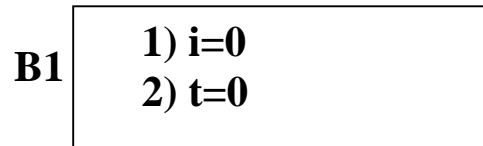
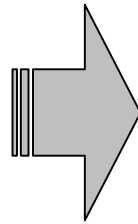
- **Set of Basic Block** : The following steps are used to set the Basic Block:
  1. **Determine the Block beginning:**
    - i- The First instruction
    - ii- Target of conditional & unconditional Jumps.
    - iii- Instruction follow Jumps.

## 2. Determine the Basic Blocks:

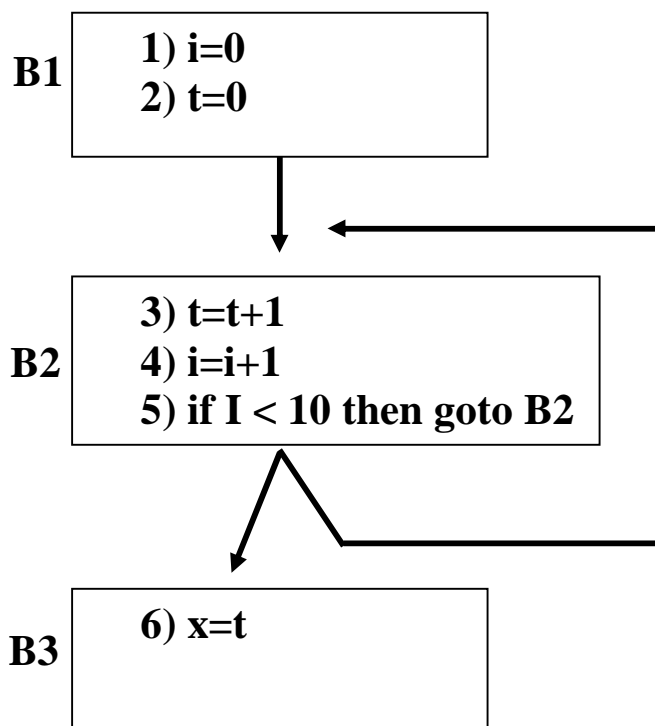
- i- There is Basic Block for each Block beginning.
- ii- The Basic Block consist of the Block beginning and runs until the next Block beginning or program end.

### Example\

- 1)  $i=0$
- 2)  $t=0$
- 3)  $t=t+1$
- 4)  $i=i+1$
- 5) if  $I < 10$  then goto 3
- 6)  $x=t$



**Basic Blocks**



**Control Flow**

## Data – Flow Analysis ( DFA )

In order to do code optimization a compiler needs to collect information about program as a whole and to distribute this information to each block in the flow graph. DFA provides information about how the execution of a program may manipulate its data , and it provides information for *global optimization* .

There are many DFA that can provide useful information for optimizing transformations. One data-flow analysis determines how definitions and uses are related to each other, another estimates what value variables might have at a given point, and so on. Most of these DFAs can be described by data flow equations derived from nodes in the flow graph.

**Reaching Definitions Analysis:** All definitions of that variable, which reach the beginning of the block, as follow:

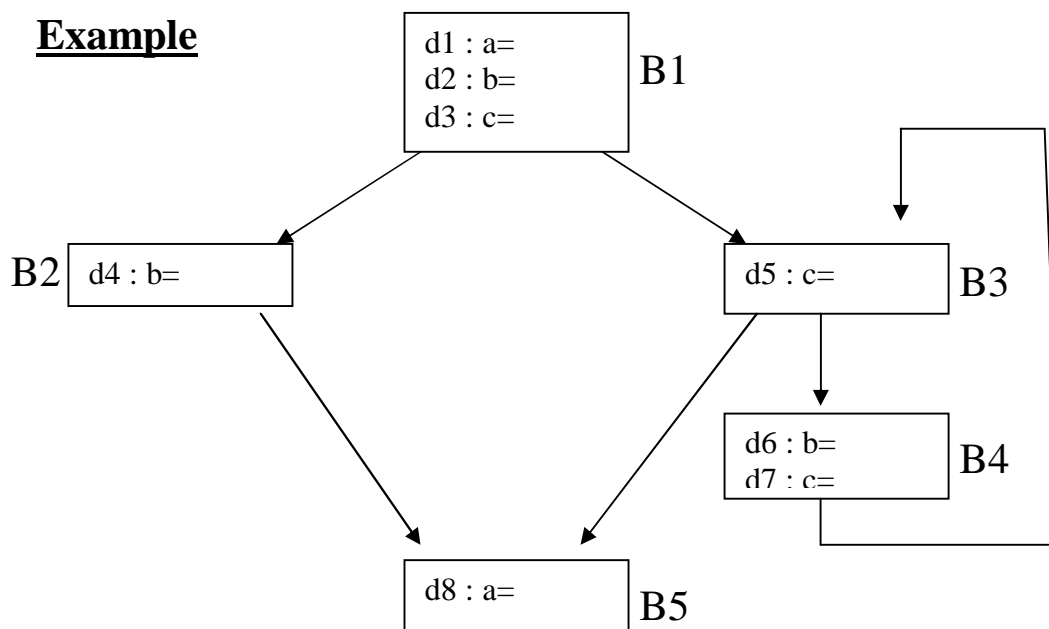
1. **Gen[B]** : contains all definitions  $d:v=e$  , in block B that  $v$  is not defined after  $d$  in B.
2. **Kill[B]** : if  $v$  is assigned in B , then Kill[B] contains all definitions  $d:v=e$ , in block different from B.
3. **In[B]** : the set of definitions reaching the beginning of B.

$$\mathbf{In[B]} = \cup \mathbf{Out[H]} \quad \text{where } H \in \text{Pred[B]}$$

4. **Out[B]** : the set of definitions reaching the end of B.

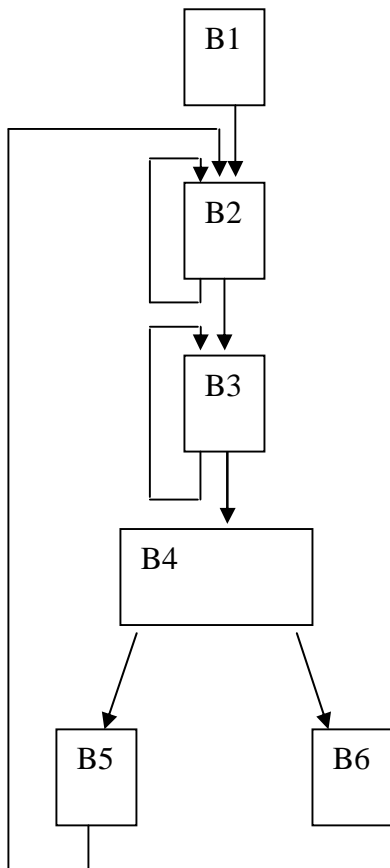
$$\mathbf{Out[B]} = \mathbf{Gen[B]} \cup ( \mathbf{In[B]} - \mathbf{Kill[B]} )$$

### Example



Block	Gen	Kill	In	Out
<b>B1</b>	d1d2d3	d4d5d6d7d8	∅	d1d2d3
<b>B2</b>	d4	d2d6	d1d2d3	d1d3d4
<b>B3</b>	d5	d3d7	d1d2d3d6d7	d1d2d5d6
<b>B4</b>	d6d7	d2d3d4d5	d1d2d5d6	d1d6d7
<b>B5</b>	d8	d1	d1d2d3d4d5d6	d2d3d4d5d6d8

**Loop Information:** The simple iterative loop which causes the repetitive execution of one or more *basic blocks* becomes the prime area in which optimization will be considered. Here we determine all the loops in program and limit *headers* & *preheaders* for every loop, for example:



**Flow Graph**

Loop No.	Header	Preheader	Blocks
1	B2	B1	2-3-4-5-2
2	B2	B1	2-2
3	B3	B2	3-3

**Loop Information**

## Code Optimization Methods

A transformation of program is called " *Local* " if it can be performed by looking only at the statements in a *Basic Block*, otherwise, it is called " *Global* " .

### Local Transformations:

1. Structure-Preserving Transformations:-

- Common Subexpression Elimination
- Dead Code Elimination

2. Algebraic Transformations:- This transformation uses to change the set of expressions ,computed by a basic block, with an algebraically equivalent set. The useful ones are those that simplify expressions or replace expensive operations by cheaper one, such as:

$$\left. \begin{array}{l} x:=x+0 \\ x:=x*1 \\ x:=x/1 \end{array} \right\} \text{ Eliminated}$$

$$x:=y^2 \implies x:=y*y$$

Another class of algebraic transformations is **Constant Folding** ,that is, we can evaluate constant expressions at compiler time and replace the constant expressions by their values, for example, the expression  $2*3.14$  would be replaced by 6.28.

### Global Transformations:

1. Common Subexpression Elimination

$$\begin{array}{l} a=b+c \\ c=b+c \\ d=b+c \end{array} \implies \begin{array}{l} a=b+c \\ c=a \\ d=b+c \end{array}$$

2. Dead Code Elimination: Variable is *dead* if never used

$$\begin{array}{l} x=y+1 \\ y=1 \\ x=2*z \end{array} \implies \begin{array}{l} y=1 \\ x=2*z \end{array}$$

### 3. Copy Propagation

<u>Origin</u>	<u>Copy Propagation</u>	<u>Dead Code</u>
$x=t3$	$x=t3$	
$a[t2]=t5$	$a[t2]=t5$	$a[t2]=t5$
$a[4]=x$	$a[4]=t3$	$a[4]=t3$
<b>Goto B2</b>	<b>Goto B2</b>	<b>Goto B2</b>

### 4. Constant Propagation

<u>Origin</u>	<u>Copy Propagation</u>	<u>Dead Code</u>
$x=3$	$x=3$	
$a[t2]=t5$	$a[t2]=t5$	$a[t2]=t5$
$a[4]=x$	$a[4]=3$	$a[4]=3$
<b>Goto B2</b>	<b>Goto B2</b>	<b>Goto B2</b>

### 5. Loop Optimization

- **Code Motion:** An important modification that decreases the amount of code in a loop is *Code Motion*. If result of expression does not change during loop( *Invariant Computation* ), can hoist its computation out of the loop.

```

For(i=0;i<n;i++)
    A[i]=a[i]+( x*x )/( y*y );

c=( x*x )/( y*y );
For(i=0;i<n;i++)
    A[i]=a[i]+c;

```





- **Strength Reduction:** Replaces expensive operations (Multiplies, Divides) by cheap ones ( Adds, Subs ). For example, suppose the following expression:

*For(i=1;i<n;i++){v=4\*i;s=s+v;} i is induction variable*

Then:

*v=0;*

*For(i=1;i<n;i++){ v=v+4; s=s+v; }*

**Induction Variable:** is a variable whose value changes by a constant amount on each loop iteration.

## Code Generation

In computer science, code generation is the process by which a compiler's code generator converts some internal representation of source code into a form( e.g., machine code)that can be readily executed by a machine.

### Issues in the Design of a Code Generator:-

- 1. Input to the Code Generator** :The input to the code generator consists of the intermediate representation of the source program(Optimized IR),together with information in ST that is used to determine the Run Time Addresses of the data objects denoted by the names in IR. Finally, the code generation phase can therefore proceed on the assumption that its input is free of the errors.
- 2. Target Programs** : The output of the code generator is the target program. The output code must be **Correct** and of **high Quality**, meaning that it should make effective use of the resources of the target machine. Like the IR ,this output may take on a variety of forms:
  - a. Absolute Machine Language** // Producing this form as output has the advantage that it can placed in a fixed location in memory and immediately executed. A small program can be compiled and executed quickly.
  - b. Relocatable Machine Language** // This form of the output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by linking-loader.
- 3. Memory Management** : Mapping names in the source program to addresses of data objects in run time memory. This process is done cooperatively by the Front-end & code generator.
- 4. Major tasks in code generation** : In addition to the basic conversion from IR into a linear sequence of machine instructions, a typical code generator tries to optimize the generated code in some way. The generator may try to use

faster instructions, use fewer instructions ,exploit available registers ,and avoid redundant computations. Tasks which are typically part of a compiler's code generation phase include:

**i. Instruction selection:** Is a compiler optimization that transforms an internal representation of program into the final compiled code(either Binary or Assembly).The quality of the generated code is determined by its Speed & Size. For example, the three address code (  $x=y+z$  ) can be translated into:

```
MOV y,R0
ADD z,R0
MOV R0,x
```

If three-address code is :

```
a=b+c
d=a+e
```

then the target code is :

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

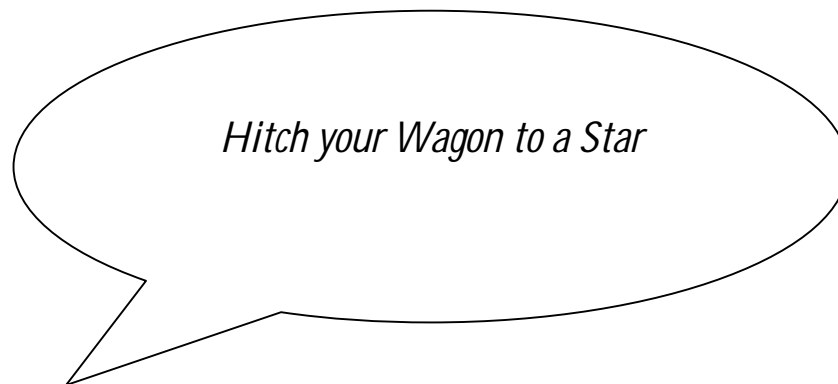
Finally, A target machine with "**Rich**" instruction set may be provide several ways of implementing a given operation. For example, if the target machine has an "increment" instruction ( **INC** ) ,then the IR  $a=a+1$  may be implemented by the single instruction ( **INC a** ) rather than by a more obvious sequence :

```
MOV a,R0
ADD #1,R0
MOV R0,a
```

**ii. Instruction Scheduling :** In which order to put those instructions. Scheduling is a *speed optimization*. The order in which computations are performed can

effect the efficiency of the target code, because some computation orders require fewer registers to hold intermediate results than others.

**iii.Register Allocation** : Is the process of multiplexing a large number of target program variables onto a small number of CPU registers. The goal is to keep as many operands as possible in registers to maximize the execution speed of software programs ( *instructions involving register operands are usually shorter and faster than those involving operands in memory* ).



## References

1. A.Aho,R.Sethi,J.D.Ullman," **Compilers- Principles, Techniques and Tools**"Addison-Weseley,2007
2. J.Tremblay,P.G.Sorenson,"**The Theory and Practice of Compiler Writing** ",McGRAW-HILL,1985
3. W.M.Waite,L.R.Carter,"**An Introduction to Compiler Construction**",Harper Collins,New york,1993
4. A.W.Appel,"**Modern Compiler Implementation in ML**" ,CambridgeUniversity Press,1998
5. Internet Papers