

جامعة الموصل
كلية التربية للعلوم الصرفة
قسم علوم الحاسوب

Compilers

Third class

اعداد

د. ميعاد محمد

1.1 Introduction:

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error-prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in.

A compiler translates (or compiles) a program written in a high-level programming language, that is suitable for human programmers, into the low-level machine language that is required by computers. During this process, the compiler will also attempt to detect and report obvious programmer mistakes. Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can detect some types of programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

Another advantage of using a high-level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler

will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

1.2 Programming Languages

A programming language is a computer language that is used by programmers (developers) to communicate with computers. It is a set of instructions written in any specific language to perform a specific task.

Hierarchy of Programming Languages based on increasing machine independence includes the following:

- 1- Machine – level languages.
- 2- Assembly languages.
- 3- High – level or user oriented languages.
- 4- Problem - oriented language.

1- Machine level language: is the lowest form of computer. Each instruction in program is represented by numeric code, and numerical addresses are used throughout the program to refer to memory location in the computers memory.

2- Assembly language: is essentially symbolic version of machine level language, each operation code is given a symbolic code such ADD for addition and MULT for multiplication.

3- A high level language such as Pascal, C.

4- A problem oriented language provides for the expression of problems in specific application or problem area. examples of such as languages are SQL for database retrieval application problem oriented language.

1.3 Translator

A translator is a program that takes as input a program written in one language and produces as output a program in another language. In addition to translating programs, the translator also plays a crucial role in error detection.

In case that the HLL specification violated, the programmers would be informed.

Important role of translator is:

1. Translating the HLL program input into an equivalent ml program.
2. Providing diagnostic messages wherever the programmer violates specification of the HLL.

1.3.1 Assembler

Programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

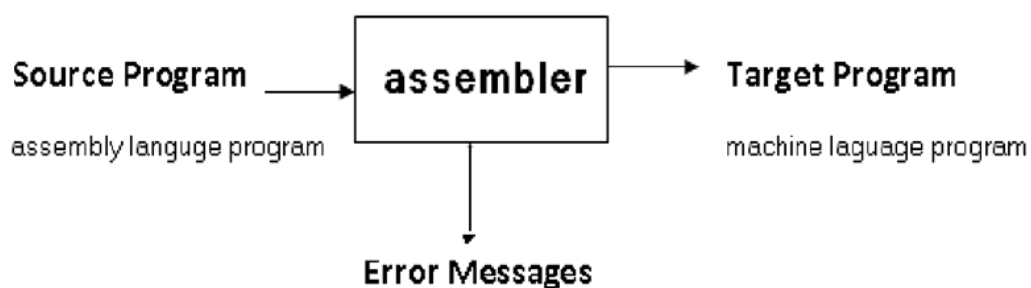
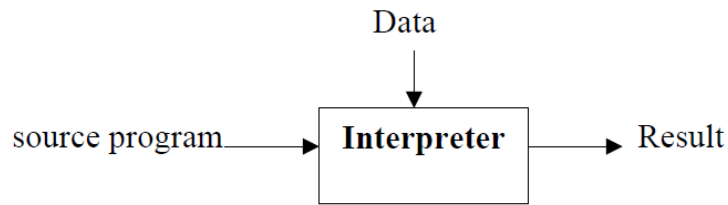


Figure (1)

1.3.2 Interpreter

An interpreter is a program that appears to execute a source program as if it were machine language. That is interpretation of the internal source from occurs at run time and an object program is generated, Fig (2) which illustrate the interpretation process.



Figure(2)

1.3.3 Compiler:

A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.

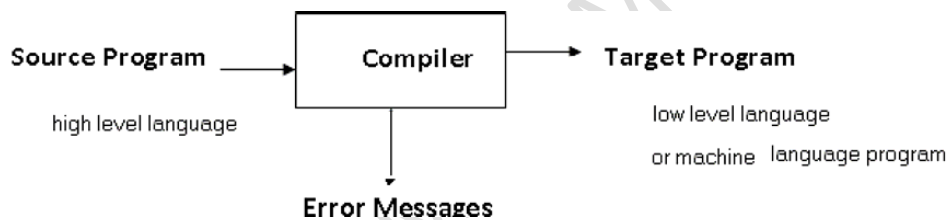


Figure (3)

The time at which the conversion of the source program to an object program occurs is called (compile time) the object program is executed at (run time). Figure (4) illustrate the compilation process Note that the program and data are processed at different times, compile time and run time respectively.

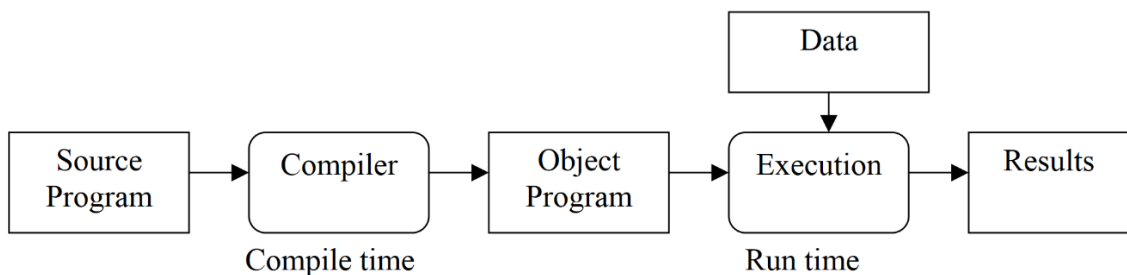


Figure (4) compilation Process

What are things that should be careful when we want to design a compiler?

1. Correctness
2. Efficiency
3. Usability

1.4 Compiler Construction Tools

Compiler construction involves various tools to aid in the development of compilers. Some common tools used in compiler construction include:

- **Lex and Yacc:** Lex is a lexical analyzer generator that converts regular expression descriptions into deterministic finite automata (DFAs) to recognize tokens in the source code. Yacc (Yet Another Compiler Compiler) is a parser generator that generates parsers based on a formal grammar specification, typically context-free grammars.
- **Bison:** Bison is a popular alternative to Yacc, providing similar functionality for generating parsers from grammar specifications.
- **Flex:** Flex is a faster alternative to Lex for generating lexical analyzers. It generates C code for scanning the input source code.

1.5 Structure of The Compiler Design

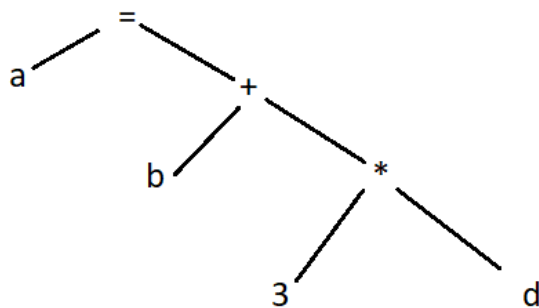
Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. There are two parts of compilation.

- a. Analysis (Machine Independent/Language Dependent)
- b. Synthesis (Machine Dependent/Language independent)

The source program is dissected into its component parts and an intermediate representation of the source program is produced during the

analysis phase. From the intermediate representation, the desired target program is constructed in the synthesis section. During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often, a special kind of tree called a syntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation.

Example: $a=b+3*d$



اسبقيات العمليات من الاعلى للأسفل

() .1

^ .2

/ , * .3

+ , - .4

= .5

Compilation process is partitioned into no-of-sub processes called '**phases**'. each of which transforms the source from one representation to another. A typical decomposition of a compiler is shown in figure (5).

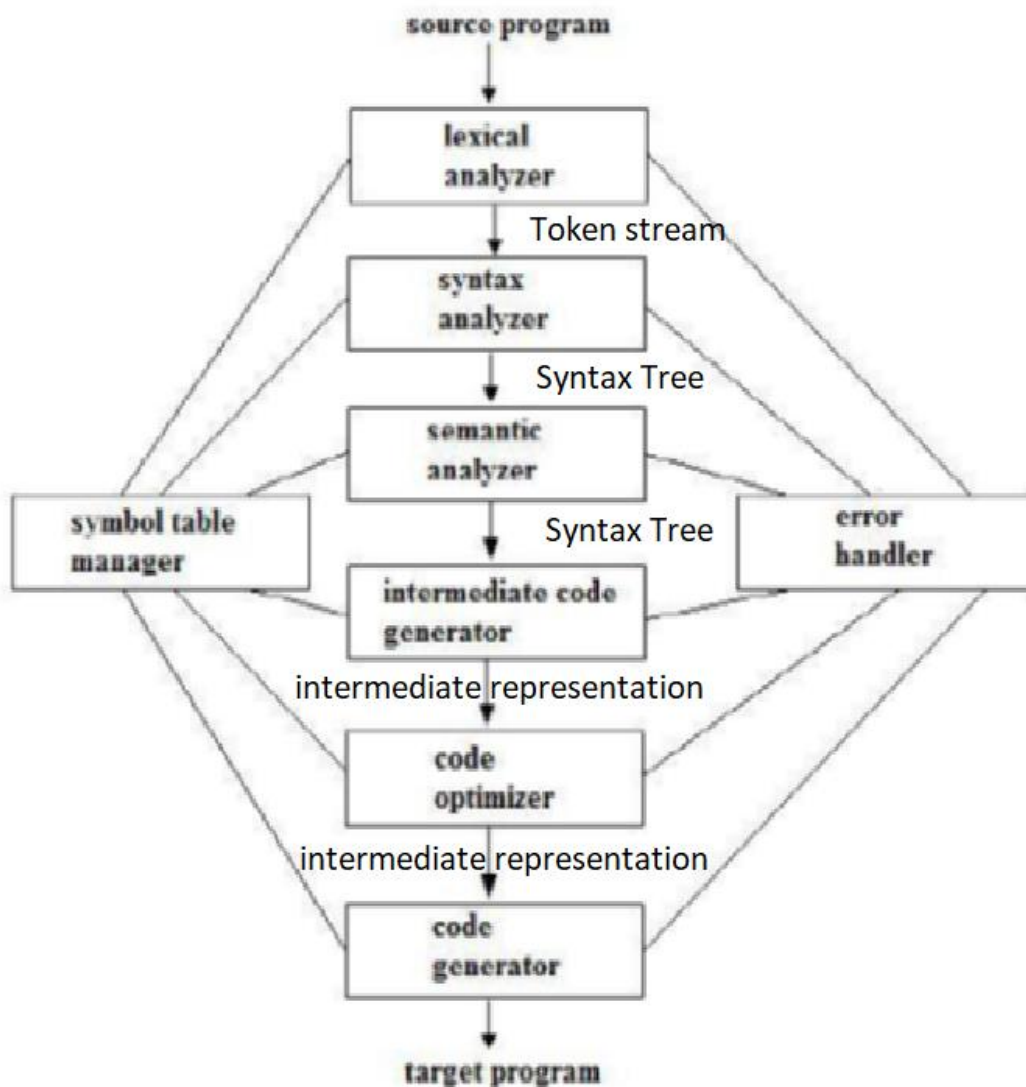


Figure (5) Compiler Phases

1. Lexical Analysis:

The lexical analyzer is the first stage of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

2. Syntax Analysis (Parsing):

The syntax analysis (or parsing) is the process of determining if a string of tokens can be generated by grammar. Every programming language has rules that prescribe the syntactic structure of well-formed programs. Syntax Analyzer takes an output of lexical analyzer and produces a parse tree.

3- Semantic analysis

The semantic analysis phase checks the source program for semantics errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements. Semantic analyzer takes the output of syntax analyzer and produces another parse tree.

4- Intermediate code generation

Generate an explicit intermediate representation of the source program. This representation should have two important properties, it should be easy to produce and easy to translate into the target program.

5- Code Optimization

Attempts to improve the intermediate code so that faster running machine code will result.

6- Code Generation

Generates a target code consisting normally of machine code or an assembly code. Memory locations are selected for each of the variables

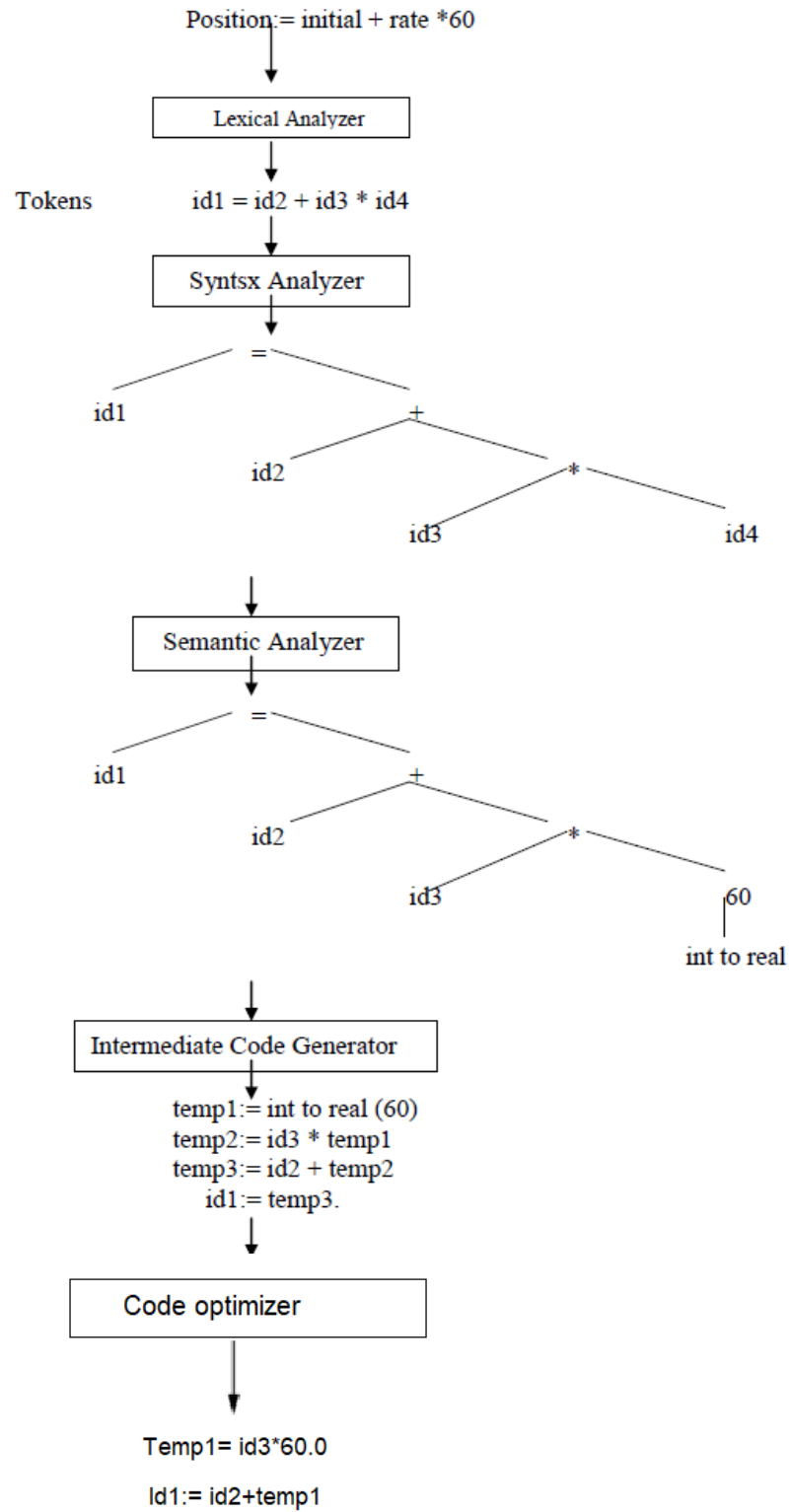
used by the program. Then intermediate instructions are each translated in to a sequence of machine instructions that perform the same task.

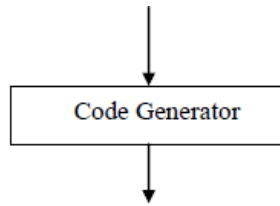
Symbol Table:

Portion of the compiler keeps tracks of the name used by the program and records essential information about each, such as type (integer, real, etc.). The data structure used to record this information is called symbolic table.

MEAAD MOHAMMED

Example:





```

MOVF id3, r2
MULF *60.0, r2
MOVF id2, r2
ADDF r2, r1
MOVF r1, id1
  
```

1.5 A simple One Pass Compiler and Multi pass Compiler:

In computer programming, a **one-pass compiler** is a compiler that passes through the parts of each compilation unit only once, immediately translating each part into its final machine code. This is in contrast to a **multi-pass compiler** which converts the program into one or more intermediate representations in steps between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.

There are many differences between one pass compiler and multi pass compiler as shown below: الفرق بين المترجم احادي المرور والمترجم متعدد المرور

One Pass Compiler	Multi Pass Compiler
1. Is a type of compiler that passes through the parts of source file only once.	1. Is a type of compiler that process source program several times.
2. All of the steps happen in one pass (read some of source file and analysis it type checked it and optimized it and generate code for it).	2. Separate compilation into multiple pass would continue with the result of previous pass.

3. Called narrow compiler because it limited scope.	3. Called wide compiler referring to the greater scope of the passes .
4. consume less memory because it don't hold intermediate representation of whole program.	4. consume much memory
5. faster	5. slower and more efficient

1.6 Error Handler:

Is called when an error in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can produced.

1.6.1 Types of Errors:

The syntax and semantic phases usually handle a large fraction of errors detected by compiler.

1. Lexical error: The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. few errors are discernible at the lexical level alone, because a lexical analyzer has a very localized view of the source program.

Example: If the string **fi** is encountered in a C program for the first time in context: **fi** (x==) , a lexical analyzer cannot tell whether **fi** is a misspelling of the keyword **if** or an undeclared function name. since **fi** is a valid identifier, the lexical analyzer must return the

token for an identifier and let some other phase of the compiler handle any error.

2- syntax error: The syntax phase can detect Errors where the token stream violates the structure rules (syntax) of the language.

Example: for i>;i=0(

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

4- runtime error : These errors occur while the program is running. They are often logical errors or conditions that arise during execution that the compiler cannot catch. Examples include dividing by zero, accessing out-of-bounds array elements, or infinity loops.

1.7 Symbol table

Symbol table (identifier or name table) is a data structure used to store information is collected by the analysis phases of the compiler. The stored information (known as attributes) are as follows (for a particular compiler):

- A- **Variable Name** اسم المتغير : must always resides in the symbol table since it is the means by which a particular variable is identified for semantic analysis & code generation. This attribute should be inserted into the symbol table during lexical analysis.
- B- Object-code address (object time address): dictates the relative location for value(s) of a variable at run time. The address is entered in the symbol table when the variable is referenced in the source

program. Insertion is performed during semantic while recalling aids in the production of correct target code for a given source program.

يحدد الموقع الخاص بقيمة المتغير في وقت التنفيذ .

C- **Type النوع**: the type of a variable is used for both – An indication of the amount of memory that must be allocated to a variable of run time (e.g. int 2 byte, char 1 byte, float 4 byte). It is the task of the semantic when handling declaration statement – for semantic checking (e.g. it is semantically inconsistent to multiply a string by a number).

D- Number-of-dimensions & Number-of-parameters :

- 1- simple variables dimension= 0,
- 2- vectors dimension= 1,
- 3- matrices dimension =2,
- 4- procedure dimension= equal to the number of parameters for that procedure.

The objective of dimension attribute is twofold:

الهدف من سمة البعد هو امران :

- 1)As a parameter in a generalized formula for calculating the address of a particular array element.

لحساب عنوان عنصر معين في المصفوفة مثل

```
int a[2];
```

```
char c;
```

إذا كان بداية العنوان هو ١٠٠ فسوف نحجز اربعة مواقع للمتغير a , إذا كان (**int = 2**)

(**byte**) فان عنوان c يبدأ بعد ١٠٤

2) For semantic checking (e.g., the number of dimensions in array reference should agree with the number specified in the declaration of the array). Also, the number of parameters in a procedure call must also agree with the number used in the procedure declaration.

عند فحص الدلالي يجب ان يكون عدد ابعاد المصفوفة عند الاشارة اليها مطابق لعدد الابعاد عند التعريف

E- Line declared: the source line number at which a variable is declared. (semantic task).

رقم السطر الذي تم تعريف المتغير عنده في البرنامج المصدر.

F- Line referenced: the source line number of all other references to the variable. (semantic task).

رقم كل سطر يشار فيه الى المتغير في البرنامج المصدر

If the last two attributes are to be included in the table then the listing of the table is called Cross-Reference Listing.

EX1/ Draw unordered symbol table that would result when compiling the following program segment (with reporting error & warning message if any).

```
1  void main();
2  {
3  int i,j[5];
4  char  c, index[5][6], block[5];
5  float  f;
6  i=0;
7  i=i+k;
8  f=f+I;
9  c='x';
10 block[4]=c;
```


11 }

الحل:

Name	Object address	Type	Dime	Line Declared	Line Referenced
i	0	int	0	3	6,7,8
j	2	int	1	3	
c	12	char	0	4	9,10
index	13	char	2	4	
block	43	char	1	4	10
f	48	float	0	5	8
k	-	-	-	-	7

Error line7: k is undefined (semantic)

Warning line8: f used before initialization (semantic)

Warning line3: j is not used (semantic)

Warning line4: index is not used (semantic)

EX2/ Draw unordered symbol table that would result when compiling the following program segment (with reporting error & warning message if any).

```

1 void main()
2 { int y,a,b;
3 char d,h[10];
4 d='z';
5 float f;
6 h[0]=d;
7 cin>>a>>b;
8 y=a+b;
9 f=y;

```

```
10 cout<<f<<y; }
```

الحل:

Name	Object address	Type	Dime	Line Declared	Line Referenced
y	0	int	0	2	8,9,10
a	2	int	0	2	7,8
b	4	int	0	2	7,8
d	6	char	0	3	4,6
h	7	char	1	3	6
f	17	float	0	5	9,10

1.7.1 Types of Symbol tables:

There are different organizations from simple which are storage efficient, to more complex which provide fast table access.

1. Unordered Symbol Table;

- simple organization
- attribute entries are added to the table in the order in which the variables are declared.
- Insertion operation requires no comparisons (other than checking for symbol table overflow).

2. Ordered Symbol Table With Binary Search;

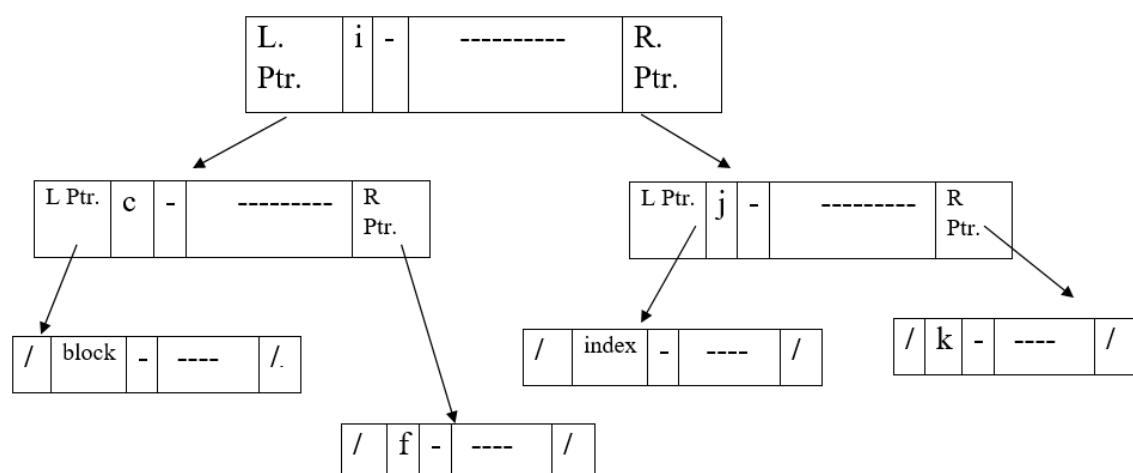
- the table lexically ordered on the variable names.
- an insertion operation must be accompanied by a lookup operation which determines where in the symbol table the variables attributes should be placed.

3. Tree Structured Symbol Tables:

Two new fields are present in a record structure, which are the left pointer and right pointer fields. Access to the tree is through the root node (top node of the tree).

Example/ draw tree structured symbol table to store the following variables: i, j, c, index, block, f, k

الحل:



4. Hash Table

A hash table can also be used to implement a symbol table, a table of constants, line numbers, etc. it can be organized as an array, or as an array of linked list, which is the method used here. We start with an array of null pointers, each of which is to become the head of a linked list. A word to be stored in the table is added to one of the lists. A **hash function** is used to determine which list the word is to be stored in. An example of a hash function would be to add the length of word to the **ASCII** code of the first letter and take the remainder on division by the array size, so that

$$\text{Hash (كلمة)} = (\text{الاسكي كود للحرف الأول من الكلمة} + \text{طول الكلمة}) \% \text{hashmax}$$

Example/ Where the variables (**frog, tree, hill, bird, bat, cat**) will be stored in a hashing type of symbol table if the hash table size is (70) records.

Hashmax= hash table size حجم الجدول

الحل:

$$\text{hash(frog)} = (4+102)\%6=4$$

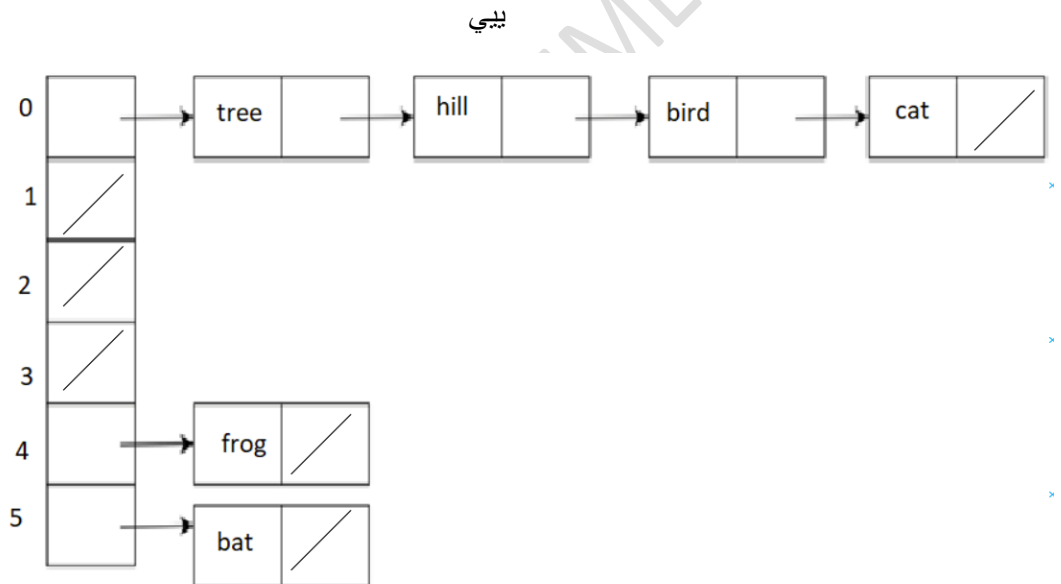
$$\text{hash(tree)} = (4+116)\%6=0$$

$$\text{hash(hill)} = (4+104)\%6=0$$

$$\text{hash(bird)} = (4+98)\%6=0$$

$$\text{hash(bat)} = (3+98)\%6=5$$

$$\text{hash(cat)} = (3+99)\%6=0$$



We required several capabilities of the symbol table we need to be able to:

- 1- Determine if a given name is in the table, the symbol table routines are concerned with saving and retrieving tokens.
- 2- Access the information associated with a given name, and add new information for a given name.
- 3- Delete a name or group of names from the table.

2. Lexical Analysis

2.1 The Role of lexical analysis :

- Is the first phase of compiler. the main task of lexical analyzer or lexer for short is to read the input characters and produce a sequence of tokens such as names, keywords, punctuation marks etc.. for syntax analyzer.
- The interaction between lexical analyzer and syntax error implemented by making lexical analysis be a subroutine of the parser. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

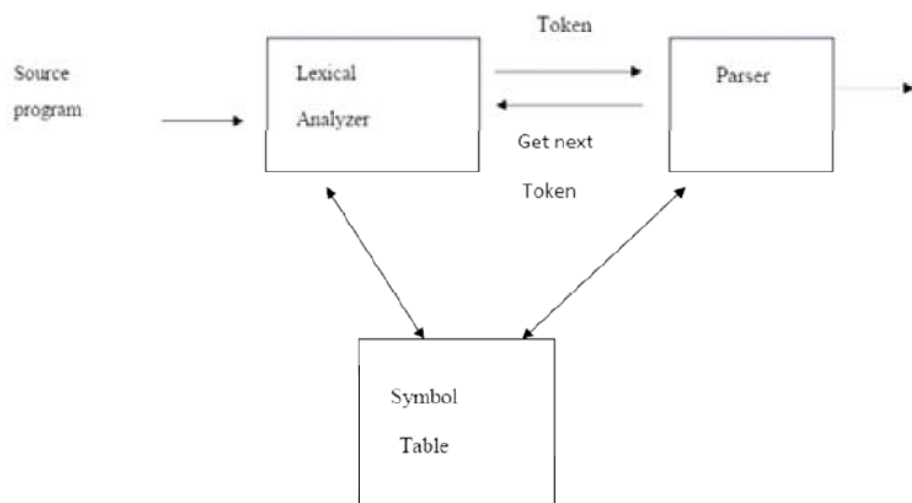


Fig. (6) Interaction of lexical analyzer with parser

The lexical analyzers is divided into a cascade of two parts: the first called scanning and the second called lexical analysis , scanner is responsible for

doing a simple task like stripping out from the source program comments and white space , while lexical analysis does the more complex tasks .

Token ,Lexeme, Pattern:

Token : a set of constructs forming the source program language like keyword, identifier, constant .

Lexeme: a sequence of characters in source program that is matched by the rule for a token.

مجموعة من الحروف التي تنطبق عليها ال rule الخاصة بالتوكن .

Pattern: the rule that describing the set of lexemes that can represent a particular token in source program .

القانون الذي يصف مجموعة من ال lexemes والتي تمثل توكن معينة ويعتبر محور عمل ال lexical analyzer . سيتم استخدام وسيلة دقيقة جدا لصياغة وصف كل توكن بشكل دقيق وهي ال RE .

RE for identifier : $id \rightarrow letter (letter | digit)^*$

في المترجمات يتم استخدام ال RE لوصف المفردة الواحدة اما لوصف جملة كاملة سنستخدم ال grammar.

مثال / جدول يوضح token ,pattern,lexeme

	token	Informal description	Sample lexeme
1	If	Character I,f	If
2	else	Character e,l,s,e	else
3	Comparison	< or <= or >= or > or !=	<=,>=,!= ...
4	Identifier	Letter followed by letter or digits	D5,pi,initial
5	number	Any numeric constant	3.452 ,56 ,120

ال lexical analyzer يكون لكل توكن مجموعة من الصفات (lexical attribute) والتي هي عبارة عن معلومات عن هذه التوكن.

Types of token:

1. Constant
2. Identifier
3. Operators
4. Relational operations
5. Special characters
6. keywords

2.2 Input Buffering:

To find tokens, the lexical analyzer goes through each character in the source program one at a time. The lexical analyzer should ideally receive its input from the buffer. The token that is being found starts at the beginning with one pointer, from the beginning pointer, a look ahead pointer scans until a token is found.

Since the lexical analyzer reads letter by letter, it is unreasonable for it to perform disk access when reading each letter, so it fetches a set of letters and stores them in a store called a buffer, and this store may have a capacity of 100 letters or 200 or more.

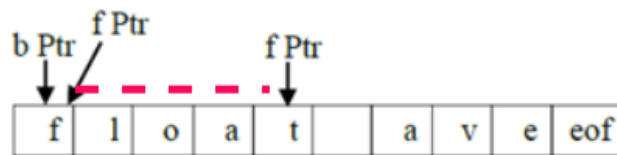
بما ان ال lexical analyzer يقرأ حرف حرف فمن غير المعقول ان يقوم بعملية الوصول للقرص (disk access) عند قراءة كل حرف لذلك يقوم بجلب مجموعة من الاحرف وتخزينها في مخزن يسمى buffer وهذا المخزن قد تكون سعته ١٠٠ حرف او ٢٠٠ او اكثر .

There are two pointers for each buffer:

هناك مؤشرين لكل buffer هما :

1. begin pointer (bptr): a pointer that indicates the beginning of the current lexeme.
2. forward pointer (fptr): a pointer that moves forward until the pattern is matched, i.e. when the set of letters that were read form a specific token.

1. bptr : مؤشر يشير الى بداية ال lexeme الحالي .
2. fptr : مؤشر يتحرك الى الامام الى ان تتم مطابقة ال pattern اي عندما تشكل مجموعة الاحرف التي تمت قراءتها تكون معينة .



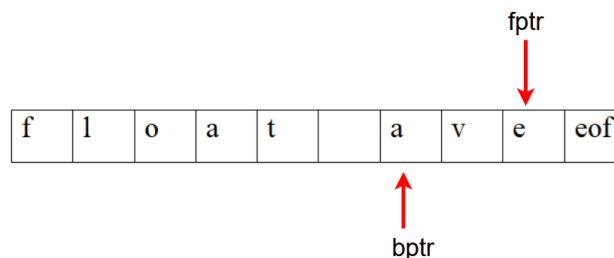
When the end of the buffer (eof) is reached, the Lexical analyzer will reload the buffer.

- Lexical analyzer either uses One Buffer or Two Buffer, but when using One Buffer, we will face a problem if the lexeme comes at the end of the buffer, and the buffer fills before the lexeme ends, as follow:

عند الوصول إلى نهاية المخزن (eof) (، Buffer) فإن ال Lexical analyzer سيقوم بعملية إعادة تحميل للمخزن ، Lexical analyzer إما يستخدم One Buffer ، أو Two Buffer لكن عند استخدام One Buffer سنواجه مشكلة إذا جاءت اللكسيم (Lexeme) في نهاية البفر ، وامتلى البفر قبل أن تنتهي اللكسيم كالاتي:

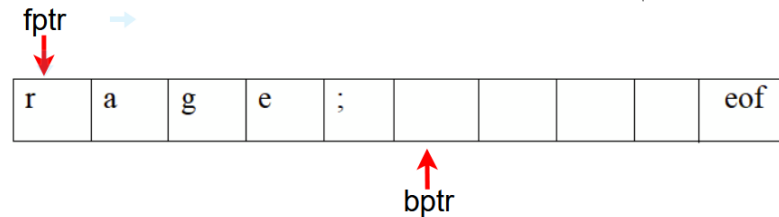
float average;

cin>>average;



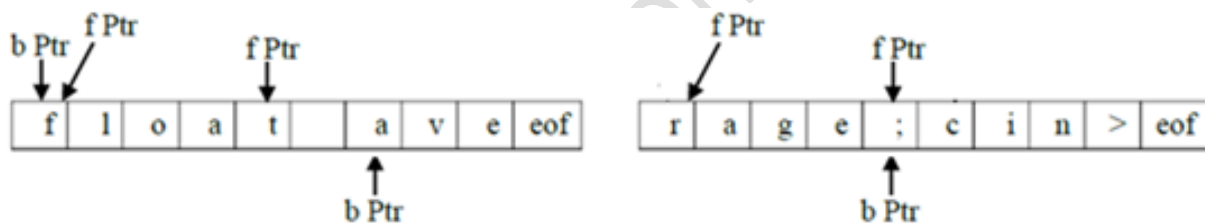
If the fptr reaches the end of the buffer and we reload it, new data will be written over the old data and we will lose the beginning of the lexeme.

إذا وصل ال fptr إلى نهاية البفر وعملنا إعادة تحميل له، فستكتب بيانات جديدة فوق البيانات القديمة وبذلك سوف نفقد بداية اللكسيم.



To overcome this problem , two buffers will be used . If we reach the end of the first buffer, the second buffer from the source program will be reloaded, and if we reach the end of the second buffer, the first buffer will be reloaded, and so on until the source program ends.

إذا وصلنا إلى نهاية البفر الأول ستجرى عملية إعادة تحميل للبفر الثاني من البرنامج المصدر، وإذا وصلنا إلى نهاية البفر الثاني ستجرى عملية إعادة تحميل للبفر الأول وهكذا الى ان ينتهي البرنامج المصدر.



When f ptr reaches the end of the lexeme, the Lexical analyzer will take the word between . bptr and fptr

عند وصول ال f ptr إلى نهاية اللكسيم فإن ال Lexical analyzer سيأخذ الكلمة المحصورة بين ال bptr و fptr.

2.3 Recognition of Token

For a programming language there are various types of tokens such as identifier, keywords, constants, and operators and so on. To represent these terms, the Lexical analyzer will send a pair of values to the parser:

<Token type, Token value>

For example –

We will consider some encoding of tokens as follows.

Token	Code	Value
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	Ptr to symbol table
constant	6	Ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	4
!=	7	5
(8	1
)	8	2
+	9	1
-	9	2
=	10	–

Symbol table

The corresponding symbol table for identifiers and constants will be,

Location Counter	Type	Value
100	identifier	a
:	:	:
105	constant	10
:	:	:
107	identifier	i
:	:	:
110	constant	2

Consider, a program code as

if (a<10)

The lexical analyzer will generate following token stream:

<1, __> <8,1> <5,100> <7,1> <6,105> <8,2>

H.W/ **i=i+2;**

else

i=i-2;

2.4.1 Specification of Token

In compiler design, Regular expressions are powerful tools for specifying and describing token patterns.

Regular Expression: Regular expressions is a useful notation suitable for describing tokens. A regular definition gives names to certain regular expressions and uses those names in other regular expressions.

Strings

A string over some alphabet is a finite sequence of symbol taken from that alphabet.

For example, banana is a sequence of six symbols (i.e., string of length six) taken from ASCII computer alphabet.

The empty string denoted by ϵ , is a special string with zero symbols (i.e., string length is 0).

If x and y are two strings, then the concatenation of x and y , written xy , is the string formed by appending y to x .

For example, If $x = \text{dog}$ and $y = \text{house}$, then $xy = \text{doghouse}$.

For empty string, ϵ , we have $S\epsilon = \epsilon S = S$.

String exponentiation concatenates a string with itself a given number of times:

$$S^2 = SS \text{ or } S.S$$

$$S^3 = SSS \text{ or } S.S.S$$

$$S^4 = SSSS \text{ or } S.S.S.S \text{ and so on}$$

By definition S^0 is an empty string, ϵ , and $S^1 = S$. For example, if $x = ba$ and na then $xy^2 = banana$.

Languages

A language is a set of strings over some fixed alphabet. The language may contain a finite or an infinite number of strings.

Let L and M be two languages where $L = \{\text{dog, ba, na}\}$ and $M = \{\text{house, ba}\}$ then

- Union: $L \cup M = \{\text{dog, ba, na, house}\}$
- Concatenation: $LM = \{\text{doghouse, dogba, bahouse, baba, nahouse, naba}\}$
- Exponentiation: $L^2 = LL$
- By definition: $L^0 = \{\epsilon\}$ and $L^1 = L$

The kleene closure of language L , denoted by L^* , is "zero or more Concatenation of L ."

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \dots \cup L^n \dots$$

For example, If $L = \{a, b\}$, then

$$L^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aba, baa, \dots\}$$

The positive closure of Language L , denoted by L^+ , is "one or more Concatenation of" L .

$$L^+ = L^1 \cup L^2 \cup L^3 \dots \cup L^n \dots$$

For example, If $L = \{a, b\}$, then

$$L^+ = \{a, b, aa, ba, bb, aaa, aba, \dots\}$$

2.5 Finite Automata: A recognizer for a language is a program that takes a string x , and answers “yes” if x is a sentence of that language, and “no” otherwise. We call the recognizer of the tokens as a **finite automaton**.

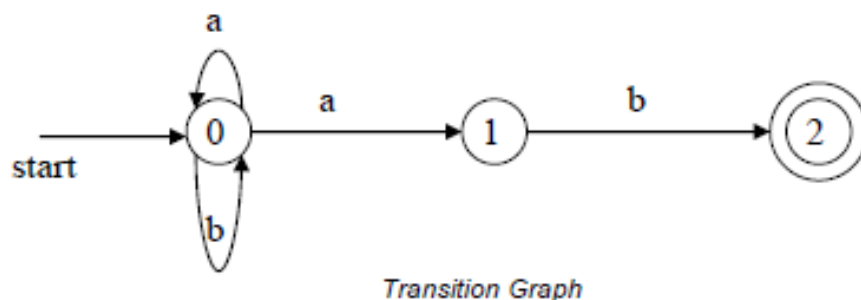
A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*

-Non-Deterministic Finite Automaton (NFA):

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
 - S - a set of states
 - Σ - a set of input symbols (alphabet)
 - move - a transition function move to map state-symbol pairs to sets of states.
 - s_0 - a start (initial) state
 - F - a set of accepting states (final states)
- Λ - transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x , if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x .

Example: Following figure shows an NFA that recognizes the language:

$(a \mid b)^* a b$



0 is the start state s_0
 $\{2\}$ is the set of final states F
 $\Sigma = \{a,b\}$
 $S = \{0,1,2\}$

Transition Function:

	a	b
0	$\{0,1\}$	$\{0\}$
1	\emptyset	$\{2\}$
2	\emptyset	\emptyset

The language recognized by this NFA is $(a|b)^*ab$

This automation is nondeterministic because when it is in state-0 and the input symbol is a , it can either go to state-1 or stay in state-0.

-Deterministic Finite Automata (DFA)

A deterministic finite automaton is a special case of a non-deterministic finite automaton (NFA) in which

1. no state has an Λ -transition
2. for each state s and input symbol a , there is at most one edge labeled a leaving s .

A DFA has at most one transition from each state on any input. It means that each entry in the transition table is a single state (as oppose to set of states in NFA).

it is very easy to determine whether a DFA accepts an input string, since there is at most one path from the start state labeled by that string.

2.6 Design Lexical Analyzer Generator:

Lexical analysis is a process of recognizing tokens from input source program. Now the question is how does lexical analyzer recognize tokens, from given source program? Well, the lexical analyzer stores the input in a buffer. It builds the regular expressions for corresponding tokens. From these regular expressions, finite automata is built. When lexeme matches with the pattern generated by finite automata, the specific token gets recognized.

There are three steps to design a lexical analyzer generator:

1. builds the regular expressions (REs) for corresponding tokens
2. build NFA for each RE
3. Convert NFA into DFA

Note: DFA: It is used to accept or reject a set of letters and symbols, so it was used for the Lexical analyzer to know whether the token belongs to this language or not.

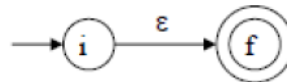
ملاحظة: DFA : تستخدم لقبول أو رفض مجموعة من الأحرف والرموز لذلك تم استخدامها لل Lexical analyzer لمعرفة هل المفردة تنتمي إلى هذه اللغة ام لا.

-Converting RE to NFA

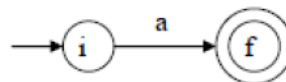
1. This is one way to convert a regular expression into a NFA.
2. There can be other ways (much efficient) for the conversion.
3. Thomson's Construction is simple and systematic method.
4. It guarantees that the resulting NFA will have exactly one final state, and one start state.
5. Construction starts from simplest parts (alphabet symbols).
6. To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA.

Rules :

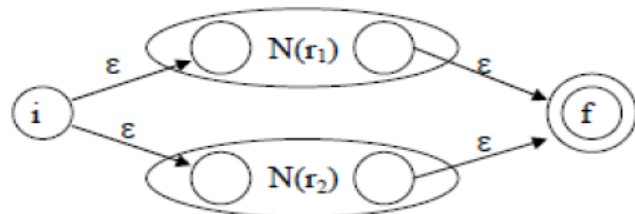
- To recognize an empty string ϵ :



- To recognize a symbol a in the alphabet Σ :



- For regular expression $r_1 \mid r_2$:



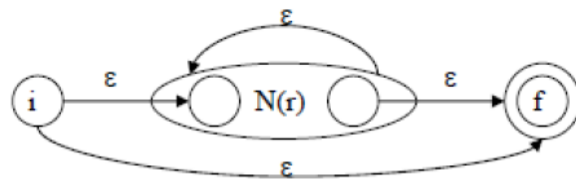
$N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2 .

- For regular expression $r_1 r_2$



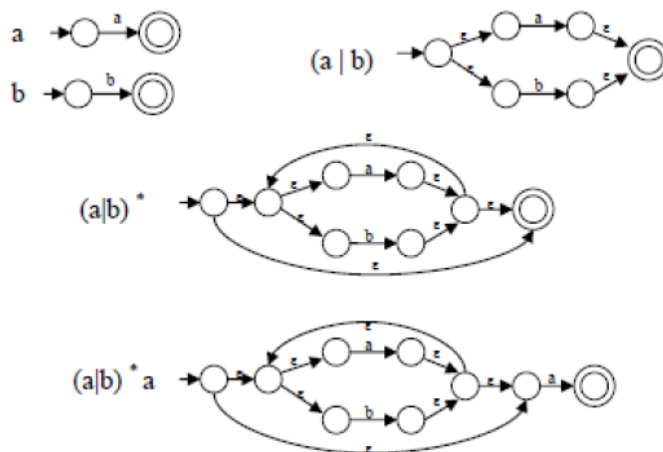
Here, final state of $N(r_1)$ becomes the final state of $N(r_1 r_2)$.

- For regular expression r^*



Example:

For a RE $(a|b)^* a$, the NFA construction is shown below.



-Converting NFA to DFA (Subset Construction)

We merge together NFA states by looking at them from the point of view of the input characters:

- From the point of view of the input, any two states that are connected by an ϵ -transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an ϵ -transition will be represented by the same states in the DFA.
- If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.

To perform this operation, let us define two functions:

- 1) The **-closure** function takes a state and returns the set of states reachable from it based on (one or more) ϵ -transitions. Note that this will always include the state itself. We should be able to get from a state to any state in its ϵ -closure without consuming any input.
- 2) The function **move** takes a state and a character, and returns the set of states reachable by one transition on this character.

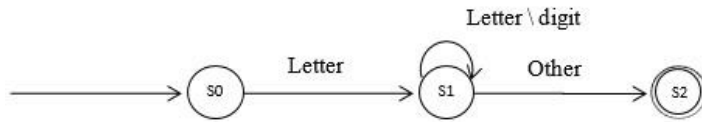
Q/ Write Regular Expression (RE) for following token and then draw transition diagram for each one:

- | | | |
|---------------|-------------------------------|-------------|
| 1- identifier | 2- Relation operation (relop) | 3- constant |
| 4- float | | |

الحل:

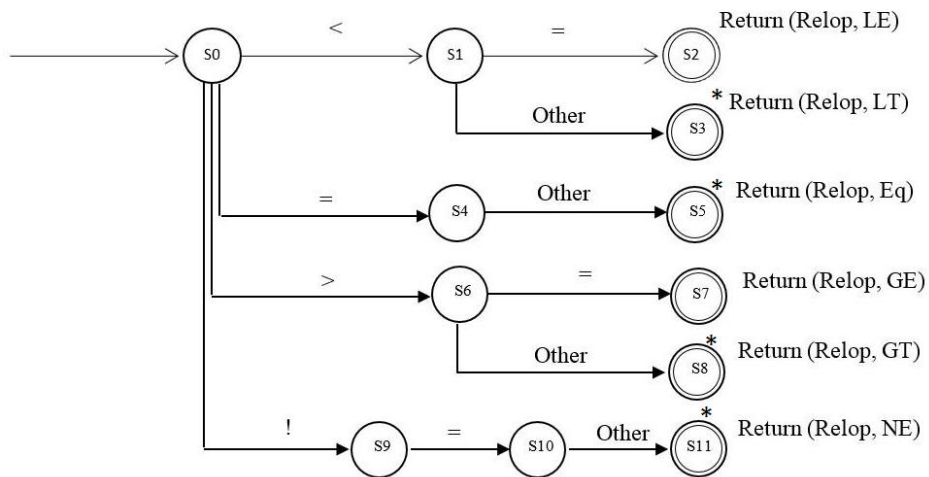
RE and TD for each token are as follow:

- | | |
|-----------------------|-----------------------|
| 1- RE of identifier : | letter(letter digit)* |
|-----------------------|-----------------------|



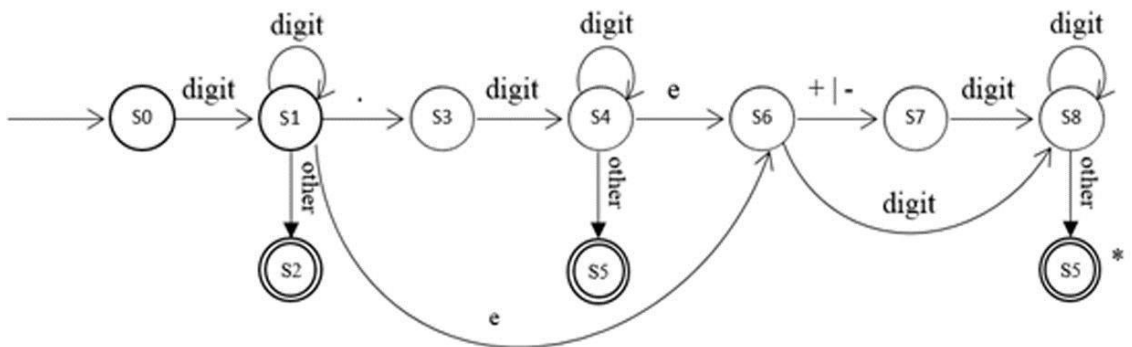
2- RE of relop:

$< | > | <= | = | != | >=$

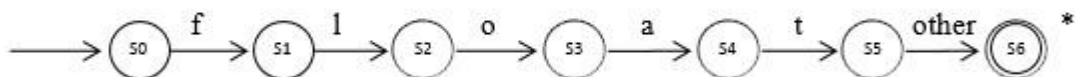


3- RE of Constant:

$(digit)^+ | (digit)^+ (.) (digit)^+ | (digit)^+ e (+|-) (digit)^+ | (digit)^+ (.) (digit)^+ e (+|-) (digit)^+ |$



4- Re of the keyword (float): float



نطبق نفس الشيء لباقي الكلمات المفتاحية مثل if , while , ... الخ

محاضرات العملي

Compilers

م.م. زيد فواز جار الله

Compiler (عملي)

Lecture (1) ((The Strings السلاسل))

ال **String السلسلة** : وهي عبارة عن نوع من أنواع البيانات تستخدم لتخزين بيانات نصية فيها, ويكون المتغير من نوع string يحتوي على مجموعة من الاحرف characters احرف تتميز بأن توضع بين علامات اقتباس (" ").

ولاستخدام البيانات من نوع string يجب تعريف header خاص بيها إضافة لل headers المكتوبة مسبقاً

```
#include <string.h>
```

وتسمى هذه بمكتبة ال string

String concatenation: وتعرف ب سلسلة ال string , وهي تجميع المتغيرات من نوع string في متغير واحد.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7      string firstName = "John ";
8      string lastName = "Smith";
9      string fullName = firstName + lastName; //or firstName + " " + lastName
10
11     cout << fullName;
12     return 0;
13 }
14
```

ملاحظة : يعتبر ال string انه object حيث يحتوي على دوال خاصة به تقوم بجميع عمليات ال string.

دالة Append: يستخدم ال string دالة append حيث تقوم هذه الدالة بعملية تشابه عملية الجمع ولكن بوجود فرق انه تضيف قيمة جديدة الى القيمة المسبقة وتسمى ب concatenation أي دمج او إضافة.

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7      string firstName = "Zaid ";
8      string lastName = "Omar";
9
10     string fullName = firstName.append(lastName); //add lastname after fullname
11
12     cout << fullName; // if we used cout<<firstName will give same output
13
14     return 0;
15 }
16

```

دوال ال String

دالة () **strlen** : تقوم هذه الدالة بارجاع قيمة عددية تمثل عدد ال characters في ال string وتمثل طول ال string .

دالة () **getlin** : وتمثل هذه الدالة عملية القراءة في ال string , تشابه دالة <<cin ولكن تتميز هذه الدالة بانها تقوم بقراءة ال space وال enter .

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7      string name;
8
9      cout << "type your full name: " ;
10     cin >> name;
11     cout << "Your name is: "<< name; // didnt take full name bcz of cin
12
13
14     return 0;
15 }
16
17

```

```

1      #include <iostream>
2      #include <string>
3      #include <string.h>
4      using namespace std;
5
6      int main()
7      {
8          char name[20];
9
10         cin.getline(name, 20);
11
12         cout<< name<<endl;
13
14         cout<<strlen (name);
15
16         return 0;
17     }
18

```

دالة النسخ (strcpy) : تستخدم هذه الدالة لعملية النسخ, وصيغتها هي strcpy(n1,n2) , حيث تقوم بنسخ n1 في n2 كما في البرنامج ادناه :

```

1      #include <iostream>
2      #include <string>
3      #include <string.h>
4      using namespace std;
5
6      int main()
7      {
8          char n1[50];
9          char n2[50];
10         cin.getline(n1, 50);
11
12         strcpy(n2, n1);
13
14         cout<<n2;
15
16         return 0;
17     }
18

```

دالة المقارنة (strcmp) : تستخدم هذه الدالة للمقارنة بين قيم ال ASCII لأول احرف char في ال string. وصيغتها هي strcmp(n1,n2) .

حيث تكون مخرجاتها بصورة ثلاثة قيم :

قيمة 0 : اذا كان ال strings متساويان بالقيمة .

قيمة موجبة 1 : اذا كان $n1 > n2$.

قيمة سالبة -1 : اذا كان $n1 < n2$.

```

1      #include <iostream>
2      #include <string>
3      #include <string.h>
4      using namespace std;
5
6
7      int main()
8      {
9          char n1[50];
10         char n2[50];
11         cin.getline(n1,50);
12         cin.getline(n2,50);
13
14         cout<<strcmp(n1,n2);
15
16         return 0;
17     }
18

```

دالة isalpha :

وتعني دالة الحروف (alphabets) , وهي دالة من دوال c++ تستخدم لفحص ال string ما اذا كان ال char السابق هو alphabet ام لا !

- ترجع قيمة ليست صفرية (non zero) اذا كان ال char السابق هو alphabet
- وترجع قيمة صفرية (zero) اذا كان ال char السابق هو not alphabet


```

1  #include <iostream>
2  #include <string.h>
3  #include <ctype.h>
4
5  using namespace std;
6
7  int main()
8  {
9      char M;
10     cin>>M;
11
12     if (isalpha(M))
13     {
14         cout<<" It's Alphabet";
15     }
16     else
17     cout << "It's Not Alphabet";
18
19     return 0;
20 }
21

```

```

1  #include <iostream>
2  #include <string.h>
3  #include <ctype.h>
4
5  using namespace std;
6
7  int main()
8  {
9
10     char str[]="AbcXYz1234567";
11
12     for (int i=0; i<strlen(str); i++)
13     {
14         if (isalpha(str[i]))
15         {
16             cout<<str[i]<<" ";
17         }
18     }
19
20     return 0;
21 }
22

```

حيث ترجع في البرنامج الثاني الحروف فقط الموجودة في ال string وتهمل أي شيء آخر سواء كان ارقام او رموز.

دالة isdigit :

وهي دالة من دوال لغة ++c تستخدم لفحص ال char الموجود في ال string ما اذا كان رقم ام لا !

- ترجع قيمة non-zero اذا كان ال char السابق هو digit
- وترجع قيمة zero اذا كان ال char السابق هو not digit

```

1  #include <iostream>
2  #include <string.h>
3  #include <ctype.h>
4
5  using namespace std;
6
7  int main()
8  {
9      char mixstr[]="abc123459xyx01";
10     for (int i=0; i<strlen(mixstr); i++)
11     {
12         if(isdigit(mixstr[i]))//using isdigit here
13         {
14             cout<<mixstr[i]<<" ";
15         }
16     }
17
18     return 0;
19 }
20
21

```

Lecture (2): Files in C++

الفايلات : وهي عبارة عن صورة تطبيقية لمخرجات البيانات التي يتنفذها المترجم , حيث يكون تمثيل البيانات بصورة ثابتة ولا تحذف بعد التنفيذ كما يحصل في البرامج التي يتنفذها المترجم في ال cmd

- تكوين الفايل يكون بحسب نوع الفايل المكتوب في ايعازات لغة c++ حيث قد يكون فايل كتابة txt او أي نوع من الفايلات الأخرى .

العمليات التي تحصل على الفايلات : كما يحصل في البيانات من عمليات Processes , يمكن للمترجم احداث عمليات على الفايلات مشابهة ولكن في حالة الفايلات تكون البيانات ثابتة ولا تتغير ولا تفقد حتى بعد إطفاء الحاسوب.

دوال الفايلات :

وهي دوال ثابتة موجودة اللغة البرمجية (c++) وتحتوي اللغة على مكتبة كاملة للدوال التي تتعامل مع الفايلات في لغة c++ .

من اهم العمليات التي تحصل على الفايلات : (mod)

1- الكتابة (Write) : عملية الكتابة على الفايلات يرمز لها بحرف "w" حيث تمثل عملية كتابة في حالة

وجود فايل معين , وفي حالة عدم وجود فايل فان هذه العملية تقوم بتكوين الفايل ومن ثم الطباعة عليه.

2- القراءة (read) : عملية القراءة من فايل مكون سابقاً ويرمز لها "r" .

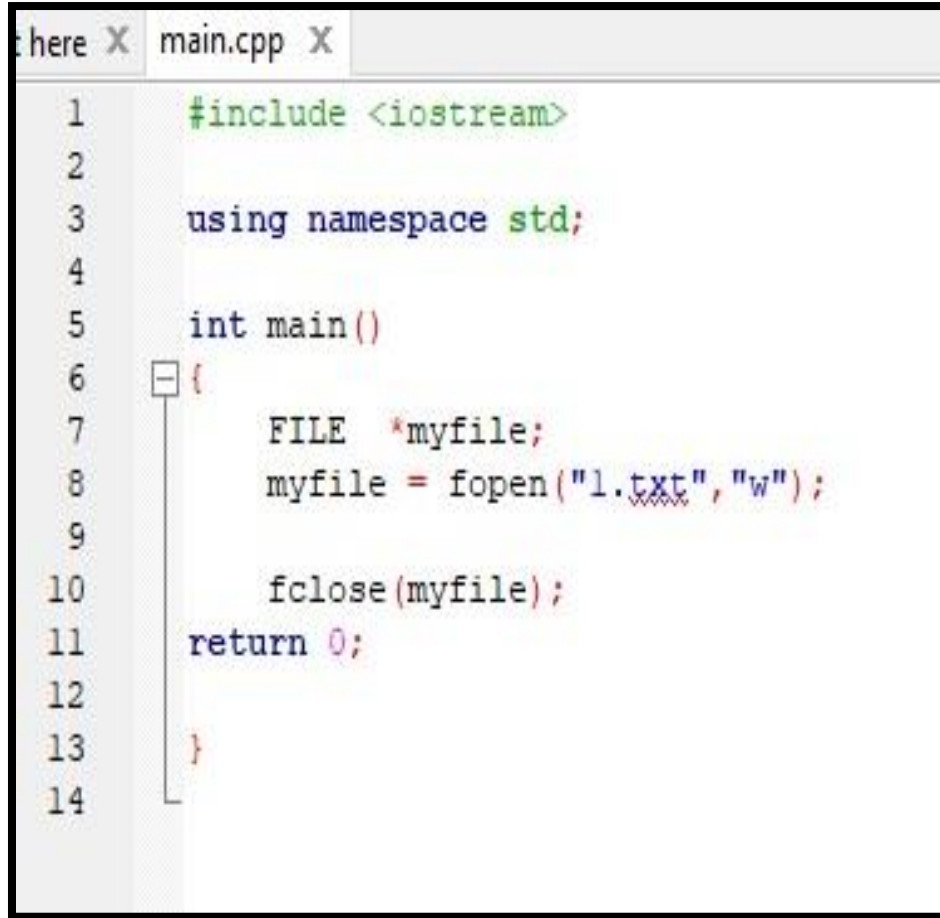
3- التعديل (append) : عملية تعديل البيانات والتي تكون موجودة مسبقاً في الفايل , ويراد إضافة بيانات الى البيانات الأولية الموجودة في الفايل.

من اهم دوال الفايلات :

دالة fopen() : تقوم هذه الدالة بتكوين وفتح الملف ويجب ان تكتب في بداية البرنامج بعد ال main() , وتحتوي على متغيرين الأول هو نوعه string يمثل مكان الفايل او الامتداد للفايل path , والمتغير الثاني يحتوي على ال mod وهي العملية التي يراد عملها على الفايل بحسب كل عملية كما مذكور مسبقاً.

صيغة كتابة دالة fopen() : fopen("path", "mod")

دالة fclose(): دالة غلق الملف , وتكتب بعد نهاية البرنامج لغرض غلق البرنامج بعد انتهاء عمل المترجم .
وتحتوي على متغير يمثل اسم الفايل المكون في دالة ال open () فقط , وصيغتها هي : fclose (filename)



```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      FILE *myfile;
8      myfile = fopen("1.txt", "w");
9
10     fclose(myfile);
11     return 0;
12 }
13
14
```

Program 1: creating first file with name (1.txt)

دالة **putc**: وتعمل هذه الدالة على ادخال البيانات على الفايل او طباعة البيانات على الفايل , تأخذ متغيرين وصيغتها هي

Putc(ch , filename)

```
1      #include <iostream>
2
3      using namespace std;
4
5      int main()
6      {
7
8
9          FILE *myfile;
10
11          char path[]="1.txt";
12          char ch;
13
14          myfile = fopen(path,"w");
15
16          cin>>ch;
17
18          putc(ch,myfile);
19
20          fclose(myfile);
21          return 0;
22      }
23
```

```

Start here X main.cpp X main.cpp X
1      #include <iostream>
2
3      using namespace std;
4
5      int main()
6      {
7
8          FILE *myfile;
9
10         char path[]="1.txt";
11         char ch='B';
12
13         myfile = fopen(path, "w");
14
15         putc(ch, myfile);
16
17         fclose(myfile);
18         return 0;
19     }
20

```

دالة cin.get: وتستخدم هذه الدالة لعملية القراءة والادخال بدلاً ل cin>> في حالة الفايلات عند استخدام الwrite.

وتتميز هذه الدالة بانها لا تهمل ال space-bar وال enter بعكس ال cin>> وحدها.

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      FILE *myfile;
8
9      char path[]="1.txt";
10     char ch;
11
12     myfile = fopen(path,"w");
13
14     do {
15         cin.get(ch);
16         //cin.get Do not neglect space or enter
17         putc(ch,myfile);
18     } while(ch!='x');
19
20
21     fclose(myfile);
22     return 0;
23 }
24
25
26

```

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      FILE *myfile;
7
8      char path[]="1.txt";
9      char ch;
10
11     myfile = fopen(path,"w");
12
13     do {
14         cin.get(ch);
15         if (ch!='x') // Do not neglect space or enter
16             putc(ch,myfile);
17     } while(ch!='x');
18
19     fclose(myfile);
20     return 0;
21 }
22
23
24
25

```

عملية الخزن في الفايلات : بالإمكان تغيير مكان الفايل واستدعاء مسار الفايل الجديد واجراء عمليات القراءة والتعديل عليه عن طريق كتابة مسار الفايل الجديد.

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      FILE *myfile;
8
9      char path[]="E:\\new\\1.txt"; // امتداد الملف يمكن تغييره الى اي ملف آخر بشرط يكتب //
10     char ch;
11
12     myfile = fopen(path,"a"); // عملية الملف append
13
14     do {
15         cin.get(ch);
16         if (ch!='x')
17             putc(ch,myfile);
18
19     } while(ch!='x');
20
21     fclose(myfile);
22     return 0;
23 }
24
25
26

```

عملية ال Append على الملفات "a" : تقوم هذه العملية بتغيير البيانات الى الملف كما تعمل ال Write ولكن الفرق انه لا تمحو هذه العملية البيانات القديمة .

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      FILE *myfile;
8
9      char path[]="E:\\new\\1.txt"; // امتداد الملف يمكن تغييره الى اي ملف آخر بشرط يكتب //
10     char ch;
11
12     myfile = fopen(path,"a"); // عملية الملف append
13
14     do {
15         cin.get(ch);
16         if (ch!='x')
17             putc(ch,myfile);
18
19     } while(ch!='x');
20
21     fclose(myfile);
22     return 0;
23 }
24
25
26

```


عملية القراءة من الفايل Read : ويرمز لها "r" وهي عملية قراءة بيانات من فايل يحتوي على بيانات سابقة .

دالة getc : تستخدم هذه الدالة في حالة قراءة البيانات عند استخدام ال read mod

مؤشر ال EOF : end of file وهو حرف مؤشر على نهاية عملية القراءة في الفايل .

امتداد الفايل path : وهو مسار لموقع خزن الفايل , ويكتب ال path كما يؤخذ في حالة ال file location ولكن بشرط تغيير ال (\) back slash ب (\\) double back slash .

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      FILE *myfile;
8
9      char path[]="E:\\new\\1.txt";
10     char ch;
11
12     myfile = fopen(path,"r");
13
14     while((ch=getc(myfile))!=EOF)
15     {
16         cout<<ch;
17     } ;
18
19
20     fclose(myfile);
21     return 0;
22
23 }
24

```