COMPILER LECTURES

COMPUTER SCIENCE

3RD CLASS

M.SC. SAMER AL-YASSIN

2017-2018

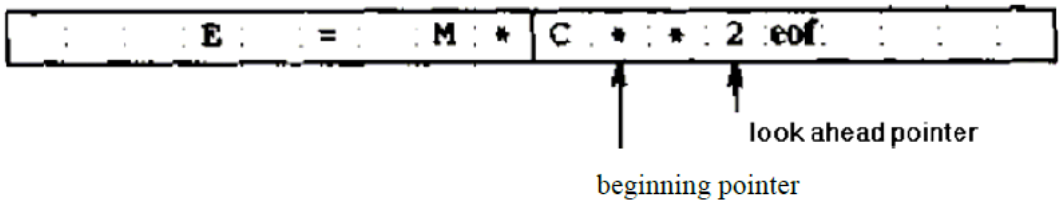
LECTURE 3

Lexical Analyzer

The lexical analyzer is the first phase of compiler. The main task of lexical analyzer is to read the input characters and produce a sequence of tokens such as names, keywords, punctuation marks etc... for syntax analyzer. This interaction summarized in fig.7, is commonly implemented by making the lexical analyzer be a subroutine of the parser. Up on receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Input buffer

Lexical analyzer scans the characters of the source program one at a time to discover tokens. It is desirable for the lexical analyzer to input from buffer. One pointer marks the beginning of the token being discovered. A look head pointer scans ahead of the beginning pointer, until a token is discovered.



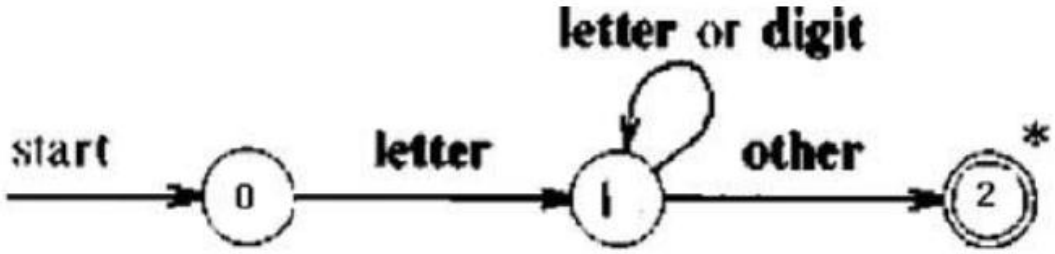
A simple approach to the design of lexical analysis :

One way to begin the design of any program is to describe the behavior of the program by a flowchart.

Remembering previous character by the position flowchart is a valuable tool, so that a specialized kind of flowchart for lexical analyzer, called ***transition diagram*** has evolved. In transition diagram, the boxes of the flowchart are drawn as circle and called ***states***. The ***states*** are connected by arrows called ***edge***. The **labels** on the various edges leaving a state indicate the input characters that can appear after that state.

To **turn** a collection of transition diagrams into a program, **we construct a segment of code for each state**. The **first step** to be done in the code for any state is to ***obtain the next character*** from the input buffer. For this purpose we use a function ***GETCHAR***, which returns the next character, advancing the look ahead pointer at each call. The **next step** is to determine which **edge**, if any out of the state **is labeled** by a character, or class of characters that includes the character just read. If no such edge is found, and the state is

not one which indicates that a token has been found (**indicated by a double circle**), we have failed to find this token. The look ahead pointer must be retracted to where the beginning pointer is, and another token must be search for using another token diagram. If all transition diagrams have been tried without success, a lexical error has been detected and an error correction routine must be called.



State 0 : C = GETCHAR ( )
if LETTER(C) then goto state1
else FAIL()
State1 : C= GETCHAR ( )
if LETTER(C) or DIGIT(C) then goto state1
else if DELIM TER(C) then goto state2
else FAIL ( )
State2: RETRACT( )
return(id,INSTALL( ))

LETTER(C) is a procedure which return true if and only if C is a letter.
DIGIT(C) is a procedure which return true if and only if C is one of the digit 0,1,…9.
DELIMITER(C) is a procedure which return true whenever C is character that could follow an identifier. The delimiter may be: blank, arithmetic or logical operator, left parenthesis, equals sign, comma,…

State2 indicates that an identifier has been found.
Since the delimiter is not part of the identifier, we must retract the look ahead pointer one character, for which we use a procedure **RETRAC**. We must install the newly found identifier in the symbol table if it is not already there , using the procedure **INSTALL**, In **state2** we return to the parser a pair consisting of integer code for an identifier, which we denoted by id, and a value that is a pointer to the symbolic table returned by **INSTALL**

Specification of Tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for set of strings.

Strings and Languages

The term of ***alphabet*** or ***character class*** denotes any finite set of symbols.
Typical examples of symbol are letter and characters. The set {0, 1} is the ***binary alphabet*** ASCII is the examples of ***computer alphabets***.

String**:** is a finite sequence of symbols taken from that alphabet. The terms ***sentence*** and ***word*** *are* often used as synonyms for term "***string***".

|S|: is the ***Length*** of the ***string*** S.

Example: |banana| =6, sequence of six symbols taken from ASCII computer alphabet.

Empty String **denoted by** (Є): special string of length 0, the string with symbols zero.

$$\epsilon S = S\epsilon = S$$

- If **x** and **y** are two strings, then the ***concatenation*** of **x** and **y**, written by **xy** is the string formed by appending **y** to **x**.
  For ***example***: if **x**=dog and **y** = house then **xy**=dog house.

Exponentiation of Strings: the string exponentiation concatenates a string with itself a given number of time.

$\mathbf{S^2}$ = SS, $\mathbf{S^3}$ = SSS, $\mathbf{S^4}$ = SSSS.

$\mathbf{S^i}$: is the string **S** repeated **i** times.

By definition $\mathbf{S^0}$ is an empty string Є.

For ***example***: if x = ba, and y=na then xy^2 =banana.

Languages: A language is any set of string formed some fixed alphabet. The language may be containing a finite or infinite number of strings.

Operations on Languages:

There are several important operations that can be applied to languages. For lexical Analysis the operations are:

1- Union.

2- Concatenation.

3- Closure.

Operation	Definition
Union L and M written $L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ in } M\}$
Concatenation of L and M written LM	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.
Positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of" L.

Example: Let L and D be two languages where L = {a, b, c} and D= {0, 1} then

1- Union: LUD = {a, b, c, 0, 1}.

2- **Concatenation**: LD = {a0, a1, b0, b1, c0, c1}.

Example: let ∑= {0, 1} and U= {000,111} and V= {101,010}.

Prove: UV≠VU

UV= {000101, 000010, 111101, 111010}

VU= {101000, 101111, 010000, 010111}

So: UV≠VU.

3- Exponentiation: L^2 = LL.

L^2= {aa,ab,ac,ba,bb,bc,ca,cb,cc}

* **By definition: L^0 = {ϵ}.**

4- Clean closure of language **L**, denoted by $\mathbf{L^*}$,"is **zero** or more **concatenation** of **L**.

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

Example: Let ***L*** = {a, b} then

L* = { ϵ , a,b , aa, ab, bb, ba , aaa, aab, aba, baa…}.

… , … , …… , ……

L^0 , L^1, L^2 , L^3 ……

5- **positive closure** of language L,denoted by $\mathbf{L^+}$,"is one or more **concatenation** of "**L**.

$L^+ = L^1 \cup L^2 \cup L^3 \ldots .\cup L^n$.

Example: Let L = {a, b} then

L^+ = { a,b , aa.ab,bb,ba ,aaa,aab,aba,baa,….}.

…. , …… , ……,…

L^1, L^2 , L^3 ……

Example: Let L and M are be two languages where L= {dog, ba, na} and M= {house, ba} then:

Union L∪M = {dog, ba, na, house}

Concatenation LM= {doghouse, dogba,bahouse,baba,nahouse,naba}

Expontention: L^2 : LL= {dogdog,dogba,dogna,baba,badog,bana,nana,nadog,naba }

H.W:

Let the alphabet ∑={a,b,c,d};U={abd,bcd};V={bcd,cab};W={da,bd}

Prove the following: 1- U (VW) = (UV) W

Regular Definitions:

A regular definition gives names to certain regular expressions and uses those names in other regular expressions.

Example1: The set of Pascal identifiers is the set of strings of letters and digits beginning with a letter.

Here is a regular definition for this set:

Letter →A| B |….| Z | a | b |….| z

Digit →0 | 1 | 2 | . . . | 9

Id →letter (letter | digit)*

The regular expression **id** is the pattern for the Pascal identifier token and defines **letter** and **digit**.

Where **letter** is a regular expression for the set of all upper-case and lower case letters in the alphabet and **digit** is the regular for the set of all decimal digits.

Example2: Unsigned numbers in Pascal are strings such as 5280, 39.37, 6.336E4, or 1.894E-4. The following regular definition provides a precise specification for this class of strings:

Digit → 0 | 1 | 2 | . . . | 9

Digits → digit digit*

Optional-fraction → . Digits | Є

Optional-exponent → (E (+ | - |) digits) | є

Num → digits optional-fraction optional-exponent

This regular definition says that

- **An optional-fraction is either a decimal point followed by one or more digits or it is missing (i.e., an empty string).**
- An optional-exponent is either an empty string or it is the letter E followed by an optional + or - sign, followed by one or more digits.

Finite Automata (FA)

It a generalized transition diagram TD, constructed to compile a regular expression RE into recognizer.

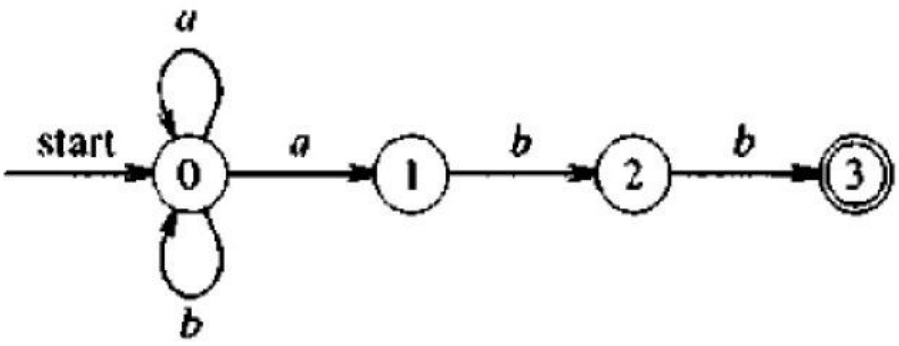
Recognizer for a Language: is a program that takes a string **X** as an input and answers "Yes" if **X** is a sentence of the language and "No" otherwise

Finite Automata FA	
Nondeterministic Finite Automata (NFA)	Deterministic Finite Automata (DFA)

Nondeterministic *Finite Automata (NFA)* :

A nondeterministic finite automation is a mathematical model consists of
1. a set of states' S;
2. a set of input symbol, Σ, called the input symbols alphabet.
3. a transition function move that maps state-symbol pairs to sets of states.
4. a state **S0** called the initial or the start state.
5. a set of states F called the accepting or final state.

An **NFA** can be described by a transition graph (labeled graph) where the nodes are states and the edges shows the transition function. The labeled on each edge is either a symbol in the set of alphabet, Σ, or denoting empty string.
Following figure shows an NFA that recognizes the language: (a | b)* a bb.



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.**

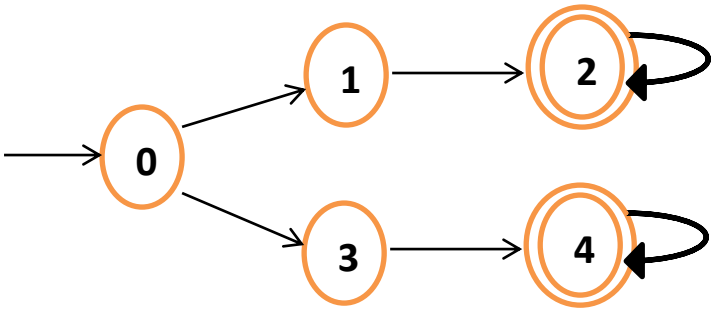
The transition is:

state	Input symbol	
	a	B
0	{0,1}	{0}
1	-	{2}
2	-	{3}

The advantage of transition table is that it provides fast access to the transitions of states and the disadvantage is that it can take up a lot of space.

Example:

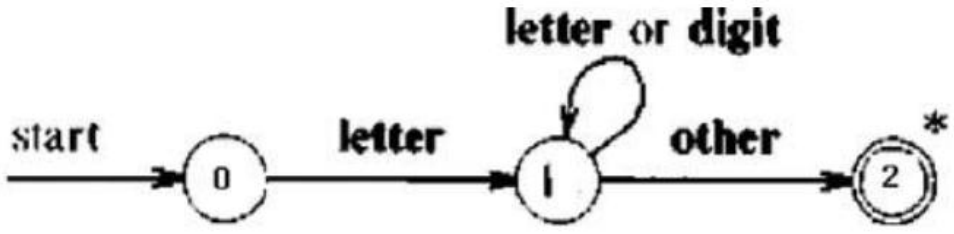
The NFA that recognizes the language **aa*|bb*** is shown below:



Example:

The following diagram shows the move made in accepting the input strings aabb :

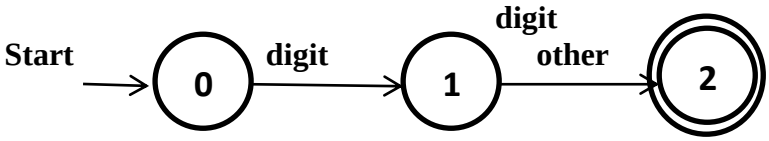
<u>Identifier</u>:



Algorithm for identifier :

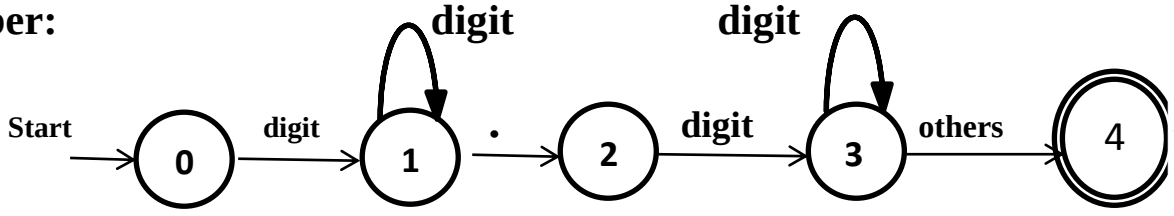
State 0: C = GETCHAR ( )
IF C = LETTER Then GOTO State 1
Else FAIL ( )
State 1: C = GETCHAR ( )
IF C = LETTER OR C = DIGIT Then GOTO State 1
Else IF C= DELIMITAR Then GOTO State 2
Else FAIL ( )
State 2: Continue ….

Integer number:



State 0: C = GETCHAR ( )
IF C = DIGIT then GOTO state1
Else FAIL ( )
State 1: C = GETCHAR ( )
IF C = DIGIT then GOTO state 1
Else IF C = DELIMTER then GOTO state 2
Else FAIL ( )
State 3: Continue ….

Real number:



Algorithm of real number :

State 0: C = GETCHAR ( )
IF C = DIGIT then GOTO state 1
Else FAIL ( )

State 1: C = GETCHAR ( )
IF C = DIGIT then GOTO state 1
Else IF C = ' . ' then GO TO state 2
Else FAIL ( )

State 2 : C = GETCHAR ( )
IF C = DIGIT then GOTO state 3
Else FAIL ( )

State 3: C = GETCHAR ( )
IF C = DIGIT then GOTO state 3
Else IF C = DELIMTER then GOTO state 4
Else FAIL ( )
State 4: Continue ….

Deterministic Finite Automata (DFA):

A deterministic finite automation (DFA, for short) is a special case of a non-deterministic finite automation (NFA) in which**:**
1. No state has an -transition, i.e., a transition on input, and

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.

Algorithm for Simulating a DFA

INPUT:

- String x
- a DFA with start state, so . . .
- a set of accepting state's F.

OUTPUT:

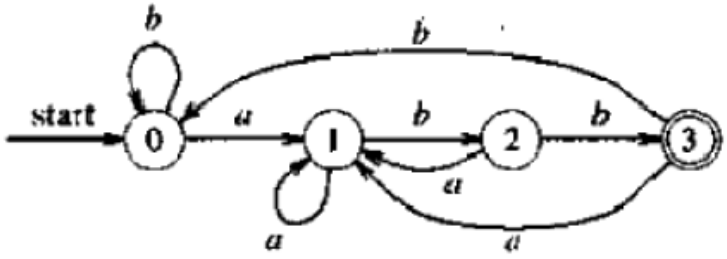
- The answer 'yes' if D accepts x; 'no' otherwise.

The function move (S, C) gives a new state from state S on input character C.

The function '**nextchar**' returns the next character in the string.

Initialization:

S := S_0
C := nextchar;
While not end – of - file do
S: = move (S, C)
C : = nextchar;
end
If S is in F then
return "yes"
else
return "No".
example :
Following figure shows a DFA that recognizes the language (a|b)*abb.



The transition table is:

state	a	b
0	1	0
1	1	2
2	1	3
3	1	0

Example of lexical analysis :

I=0
For I = 0 to 10
Sum = sum + i
Next I

keyword	identifier	Integer number	Mathematic Op	Relational Op
For	i	0	+	=
To		1		
Next		10		