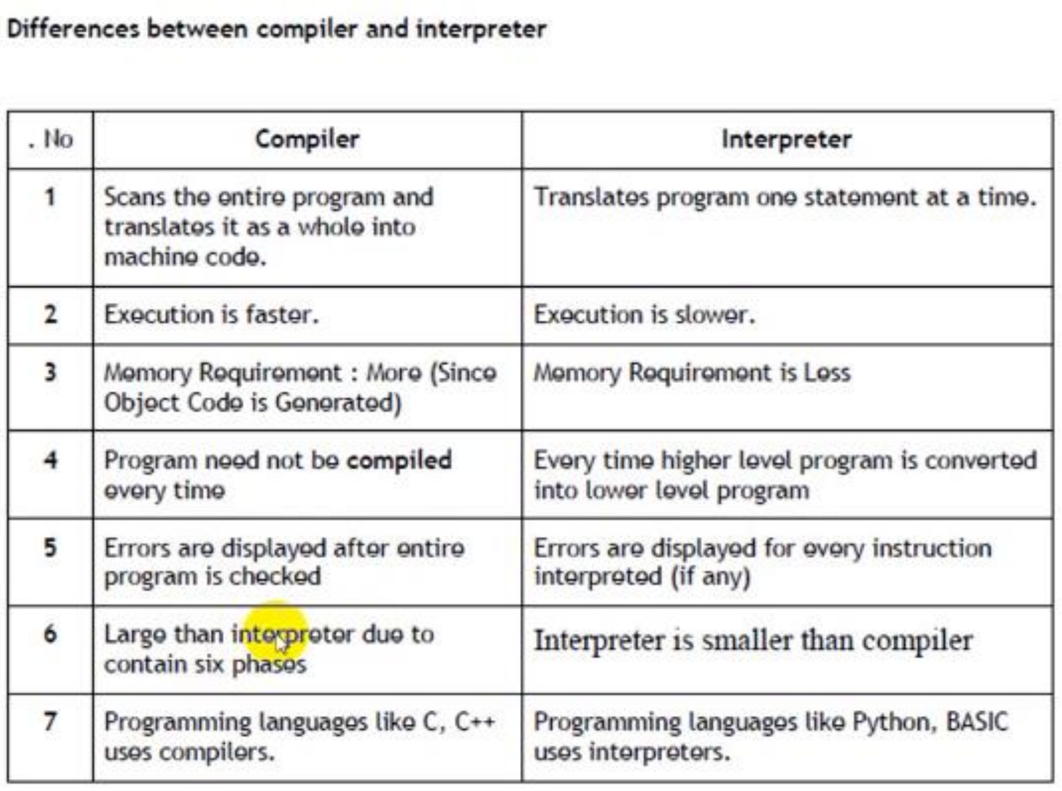
Lec Two

Difference between compiler and interpreter



Differences between compiler and interpreter

. No	Compiler	Interpreter
1	Scans the entire program and translates it as a whole into machine code.	Translates program one statement at a time.
2	Execution is faster.	Execution is slower.
3	Memory Requirement : More (Since Object Code is Generated)	Memory Requirement is Less
4	Program need not be compiled every time	Every time higher level program is converted into lower level program
5	Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted (if any)
6	Large than interpreter due to contain six phases	Interpreter is smaller than compiler
7	Programming languages like C, C++ uses compilers.	Programming languages like Python, BASIC uses interpreters.

❖ Difference between Low-Level & High-Level Language

	H.L.L	L.L.l
1. Learning التعلم	High-level languages are easy to learn. سهلة التعلم	Low-level languages are difficult to learn. صعبة التعلم
2.Understanding الفهم	High-level languages are near to human languages. قريبة من لغة الانسان	Low-level languages are far from human languages. بعيد عن لغة الانسان

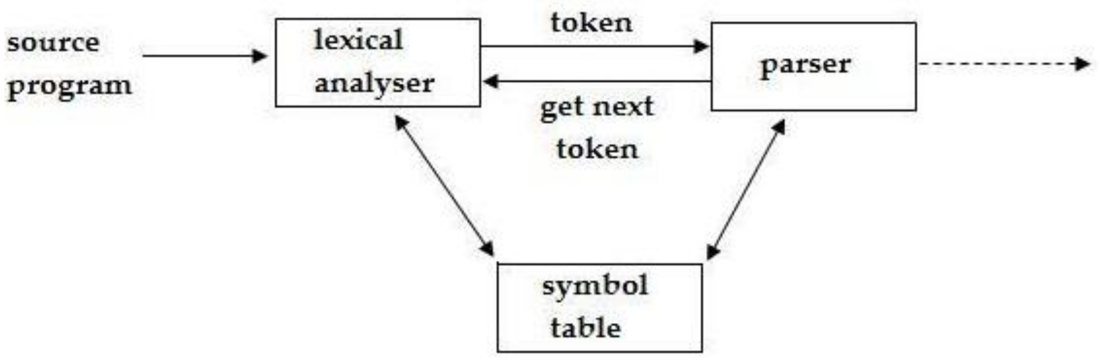
3. Execution التنفيذ	Programs in high-level languages are slow in execution. البرنامج تنفيذه بطيء	Programs in low-level languages are fast in execution. البرنامج تنفيذه سريع
4. Modification التحديث	Programs in high-level languages are easy to modify. البرنامج سهل التحديث	Programs in low-level languages are difficult to modify. البرنامج صعب التحديث
5. Facility at hardware level السهولة	High-level languages do not provide much facility at hardware level. لغات ذات المستوى العالي لا تمنح سهولة كافية	Low-level languages provide facility to write programs at hardware level. تمنح سهولة كافية
6. Knowledge of hardware Deep معرفة الأجهزة بعمق	Knowledge of hardware is not required to write programs لا تتطلب معرفة عميق بكتابة البرنامج	Deep knowledge of hardware is required to write programs. تتطلب معرفة عميقة بكتابة البرنامج
7. Uses الاستخدام	These languages are normally used to write application programs. هذه اللغات تستخدم طبيعيا لكتابة التطبيقات البرمجية	These languages are normally used to write hardware programs. هذه اللغات تستخدم طبيعيا لكتابة البرامج المادية

(Lexical Analyzer)

The Role of the Lexical Analyzer

Lexical Analyzer is the interface between the source program and the compiler. The ***main task*** of lexical Analyzer is to read the input characters and produce a sequence of tokens that the parser uses for syntax analysis.

المحلّل المعجمي الوصلةُ بين مصدر البرنامج والمفسر. إنّ المهمّةَ الرئيسيةَ للمحلّلِ المعجميِ أَنْ تَقْرأَ الرموز وتقدم على شكل سلسله من ال Token وتُنتجُ a سلسلة الرموزِ التي يَستعملُ المُعْرب اللُغوي لي syntax analysis.



Upon receiving a "get next token" command from the parser, the Lexical Analyzer reads input characters until it can identify the next token.

The Secondary Tasks of Lexical Analyzer:

1. Removal of white space and the comments. White space (blanks, tabs, and new line characters).
2. Correlating error messages from the compiler with the source program. For example, the lexical analyzer may keep track of the number of newline characters seen, so that a line number can be associated with an error message.

Input Buffering:

The lexical analyzer scans the characters of the source program one at a time to discover tokens; it is desirable for the lexical analyzer to read its input from an input buffer.

We have two pointers one marks to the beginning of the token begin discovered. A look-ahead pointer scans a head of the beginning point, until the token is discovered.

Example: if we have the statement For i:=1 To 10 Do then the buffer will be

F	O	R		i	:	=	1		T	O		10		D	O

Example: Let the following segment of source program is input to lexical analysis:

If A>=100 **Then**

  Begin

    X := y1+5.6;

   Count := A*4;

End;

Tokens Table		
Token	***Type***	***Index***
If	**Keyword**	
A	**Identifier**	**1**
>=	**Relation operator**	
100	**Constant**	**1**
Then	**Keyword**	
Begin	**Keyword**	

X	**Identifier**	**2**
:=	**Assignment operator**	
y1	**Identifier**	**3**
+	**Operation operator**	
5.6	**Constant**	**2**
;	**Punctuation**	
Count	**Identifier**	**4**
:=	**Assignment operator**	
A	**Identifier**	**1**
*****	**Operation operator**	
4	**Constant**	**3**
;	**Punctuation**	
End	**Keyword**	
;	**Punctuation**	
	Constant	
Index	Value	
1	100	
2	5.6	

3	4	

Identifier	
Index	Name
1	A
2	X
3	y1
4	Count

Note: These tables in above are saving in storage structure which called ***Symbol Table***.

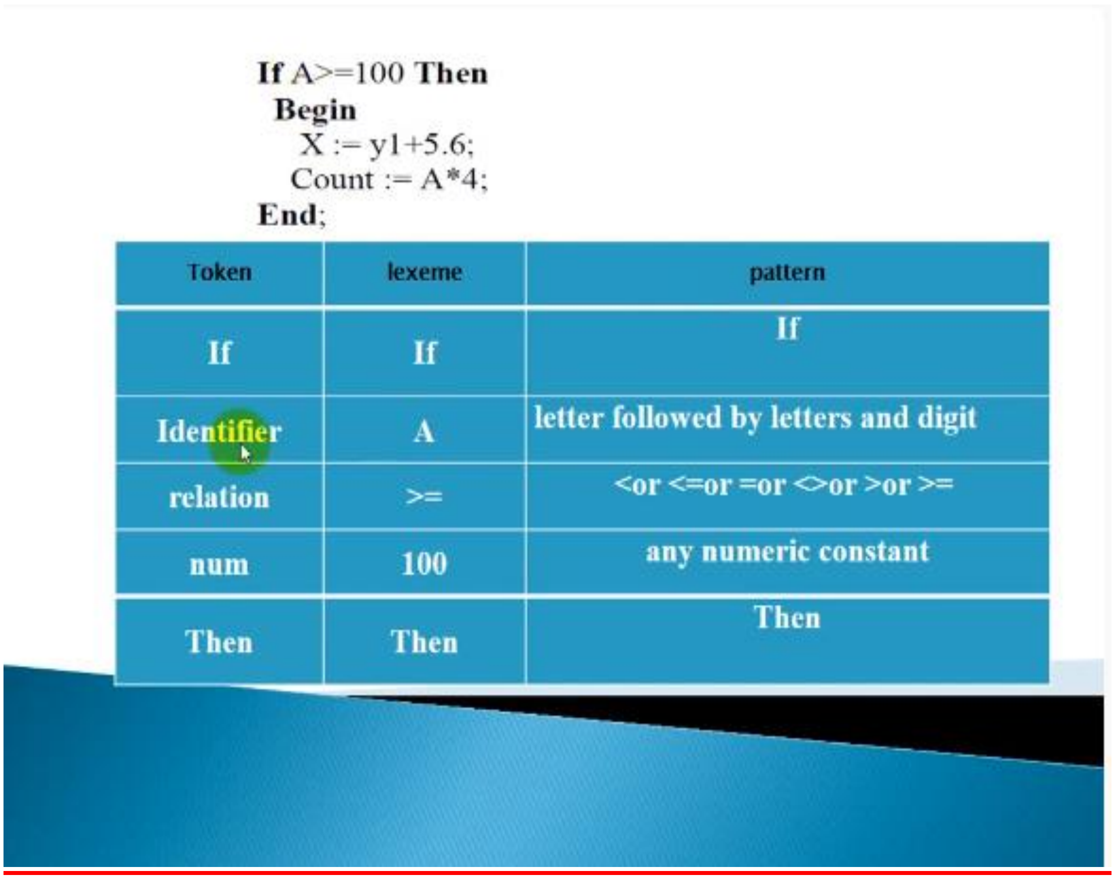
Tokens, Patterns, Lexemes

When talking about lexical analysis, we use the terms "Tokens"," Patterns", and "Lexemes" with specific meanings. Examples of their use are shown in figure below:

Token	Sample lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<,<=,=,<>,>,>=	<or <=or =or <>or >or >=

id	pi, count, d2	letter followed by letters or digit
num	3.14, 0, 45, -7.5	any numeric constant
literal	"computer"	any characters between "and" except"

Example:



```
If A>=100 Then
 Begin
   X := y1+5.6;
   Count := A*4;
End;
```

Token	lexeme	pattern
If	If	If
Identifier	A	letter followed by letters and digit
relation	>=	<or <=or =or <>or >or >=
num	100	any numeric constant
Then	Then	Then

A **lexeme** is a sequence of characters in the source program that is matched by the pattern for a token.

For example, the pattern for the Relation Operator (RELOP) token contains six lexemes (=, < >, <, < =, >, >=) so the lexical analyzer should return a RELOP token to parser whenever it sees any one of the six.

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

By using **Regular Expressions**, we can specify patterns to lexical that allow it to scan and match strings in the input. For example, the pattern for the Pascal **Identifier** token "Id" is:

letter (letter | digit)*

Example: In Pascal statement

Const Pi=3.1416;

The substring **Pi** is a lexeme for the token "Identifier".

Strings and Languages

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

Empty String (ϵ): special string of length zero.

Exponentiation of Strings

S^2 = SS S^3 = SSS S^4 = SSSS

S^i is the string **S** repeated **i** times.

By definition S^0 is an empty string.

Languages

A language is any set of string formed some fixed alphabet.

Operations on Languages

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

Union.

Concatenation.

Closure.

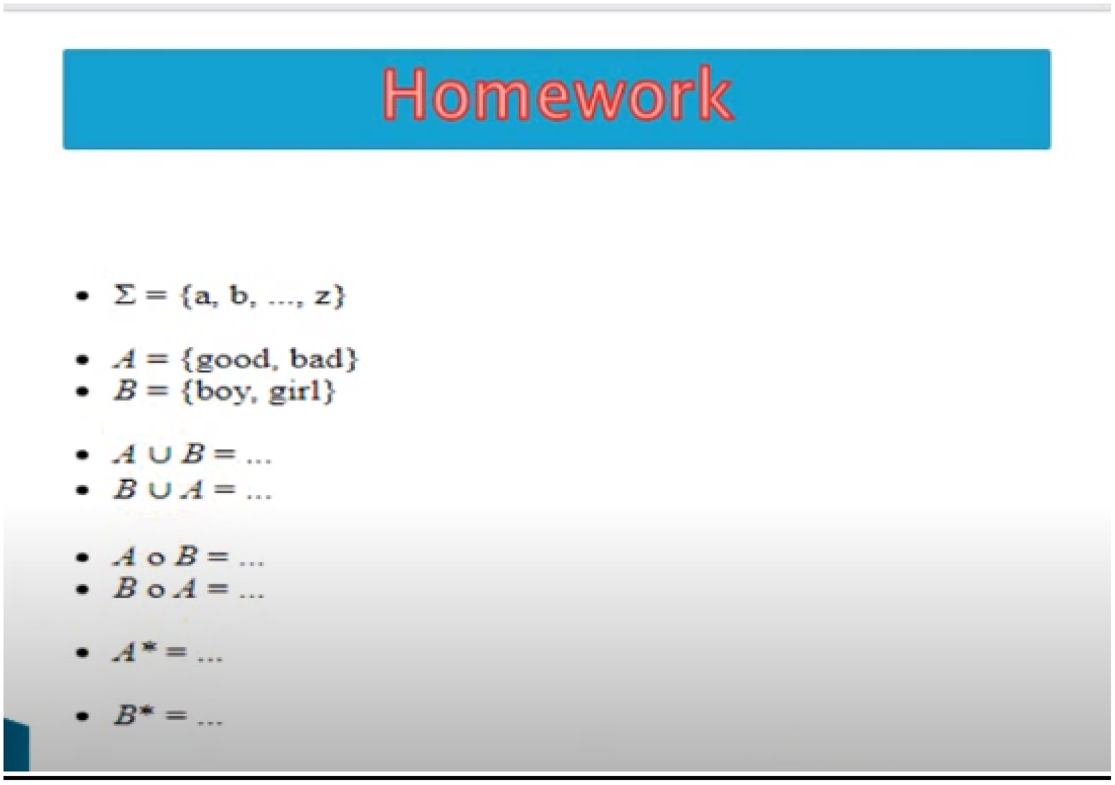
Operation	**Definition**
Union ***L*** and ***M*** written ***L*** ***M***	***L*** ***M***={s | s is in ***L*** or s in ***M***}
Concatenation of ***L*** and ***M*** written ***LM***	***LM***={st | s is in ***L*** and t is in ***M***}
Kleene closure of ***L*** written L^*	L^*= L^* denotes "zero or more concatenations of" ***L.***
Positive closure of ***L*** written L^+	L^+= L^+ denotes "one or more concatenations of" ***L***.

Example: Let ***L*** and ***M*** be two languages where ***L*** = {a, b, c} and

D= {0, 1} then

- Union: ***LUD*** = {a, b, c, 0,1}
- Concatenation: ***LD*** = {a0,a1, b0, b1, c0,c1}

- Expontentiation: $\boldsymbol{L}^2$ = LL
- By definition: $\boldsymbol{L}^0$ ={∈}



Homework

- $\Sigma = \{a, b, ..., z\}$
- $A = \{good, bad\}$
- $B = \{boy, girl\}$
- $A \cup B = ...$
- $B \cup A = ...$
- $A \circ B = ...$
- $B \circ A = ...$
- $A^* = ...$
- $B^* = ...$