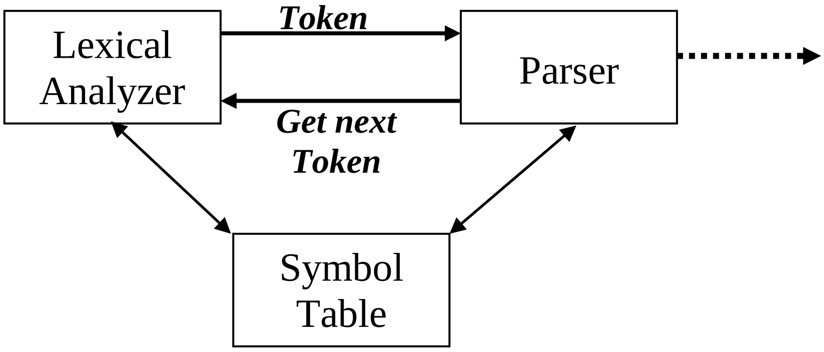
Lexical Analyzer

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.



Interaction of lexical analyzer with parser

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 newline 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.

Note: Regular Expressions are used to define the tokens recognized by lexical analyzer. The lexical analyzer is implemented as Finite Automata (DFA).

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**	

Identifier	
Index	Name
1	A
2	X
3	y1
4	Count

Constant	
Index	Value
1	100
2	5.6
3	4

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 and digit
num	3.14, 0, 45, -7.5	any numeric constant
literal	"computer"	any characters between "and" except"

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".

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 lookahead 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

↑ Beginning Pointer (at F)   ↑ Lookahead Pointer (at R)

Symbol Tables

Gather information about names and constants which are in a program. Symbol table management refers to the symbol table's storage structure, its construction in the analysis phase and its use during the whole compilation.

1) A symbol table is a data structure, where information about program objects is gathered.
2) Is used in all phases of compiler.
3) The symbol table is built up during the lexical and syntactic analysis.
4) Help for other phases during compilation:
   - ☒ **Semantic analysis**: type conflict?
   - ☒ **Code generation**: how much and what type of run-time space is to be allocated?
   - ☒ **Error handling**: Has the error message "Variable A undefined"

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

Empty String (∈): 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:

1- **Union.**

2- **Concatenation.**

3- **Closure.**

Operation	Definition
Union ***L*** and ***M*** written $L \cup M$	$L \cup M$={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^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ 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***.

<u>Example:</u> 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: L^2 = LL
- By definition: L^0 ={∈}

Regular Expressions

In Pascal, an identifier is a letter followed by zero or more letters or digits; in this section presents a notation called Regular Expressions (RE) that allows us to define precisely sets. With this notation, we might define Pascal identifiers as:

Letter (Letter | Digit)*

Vertical bar **|** means "or"

Examples: Let ∑= {a, b}

1. The RE **a | b** denotes the set {a, b}
2. The RE **(a | b) (a | b)** denotes {aa, ab, ba, bb}
3. The RE **a*** denotes {∈, a, aa, aaa, aaaa, …….}
4. The RE **(a | b)*** denotes {∈, a, b,ab,ba, bba, aaba, ababa, bb,...}
5. The RE **a | ba*** denotes the set of strings consisting of either signal **a** or **b** followed by zero or more **a's.**
6. The RE **a*****ba*****ba*****ba*** denotes the set of strings consisting exactly three **b's** in total.
7. The RE **(a | b)*****a(a | b)*****a(a | b)*****a(a | b)*** denotes the set of strings that have at least three **a's** in them.
8. The RE **(a | b)*** **(aa | bb)** denotes the set of strings that end in a double letter.
9. The RE ∈ **| a | b | (a | b)**^{3} **(a | b)*** denotes to all strings whose length is not two, could be zero, one, three, …..

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 | ∊
optional-exponent → (E (+ | - | ∊) **digits) |** ∊
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.