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 . So, when it scans the source program, it will be able to return a suitable token whenever it encounters a token lexeme. (Lexeme refers to the sequence of characters in the source program that is matched by language's character patterns that specify identifiers, operators, keywords, delimiters, punctuation symbols, and so forth.) Therefore, the lexical analyzer design must:

1. Specify the token of the language, and
2. Suitably recognize the tokens.

We cannot specify the language tokens by enumerating each and every identifier, operator, keyword, delimiter, and punctuation symbol; our specification would end up spanning several pages—and perhaps never end, especially for those languages that do not limit the number of characters that an identifier can have. Therefore, token specification should be generated by specifying the rules that govern the way that the language's alphabet symbols can be combined, so that the result of the combination will be a token of that language's identifiers, operators, and keywords. This requires the use of suitable language-specific notation.

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 conductive 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 :

< numericliteal , " *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* " > < lparent , " (" >

< assign - op , " = " >
< numlit , " 2 " >
< addop , " + " >
< ident ," f ">
< 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*** .

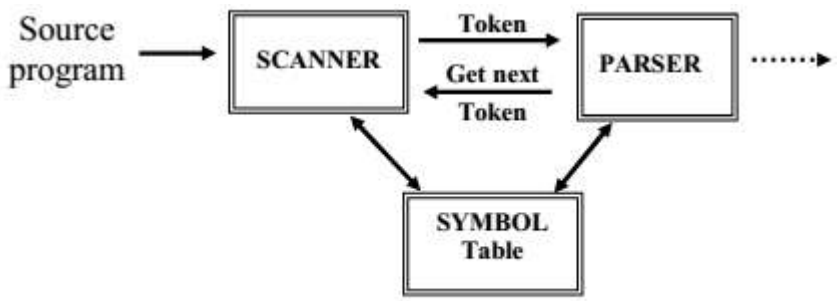
The separation of ***scanner*** and ***parser*** can have other advantages, scanning characters is typically slow in compilers and separating it from parsing particular emphasis can be given to making the process efficient .

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 does 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 cannot 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 :

Token	Internal represent	Location
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