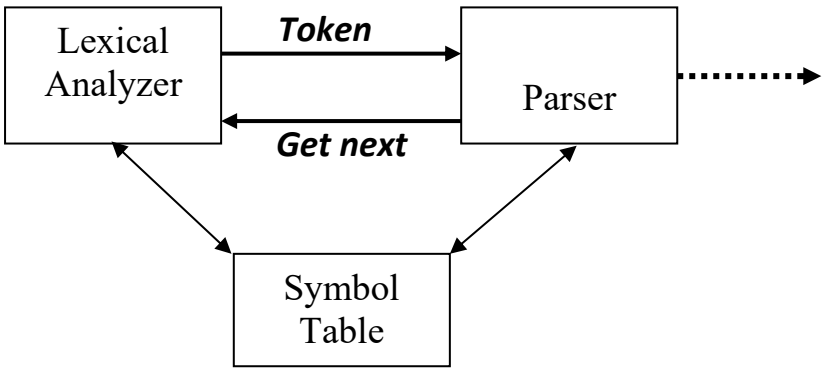
Lexical Analyzer

Introduction

1. To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
2. Secondly, having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

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 (FA).***

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

```
If (a>=100)
  {
    x = y1+5.6;
   count = a×4;
}
```

Identifier	
Index	Name
1	a
2	X
3	y_1
4	count

Constant	
Index	Value
1	100
2	5.6
3	4

Tokens Table		
Token	***Type***	***Index***
If	**Keyword**	
(	**Punctuation**	
a	**Identifier**	**1**
>=	**Relation operator**	
100	**Constant**	**1**
)	**Punctuation**	
{	**Punctuation**	
x	**Identifier**	**2**
=	**Assignment operator**	
y_1	**Identifier**	**3**
+	**Operation operator**	
5.6	**Constant**	**2**
;	**Punctuation**	
count	**Identifier**	**4**
=	**Assignment operator**	
a	**Identifier**	**1**
×	**Operation operator**	
4	**Constant**	**3**
;	**Punctuation**	
}	**Punctuation**	

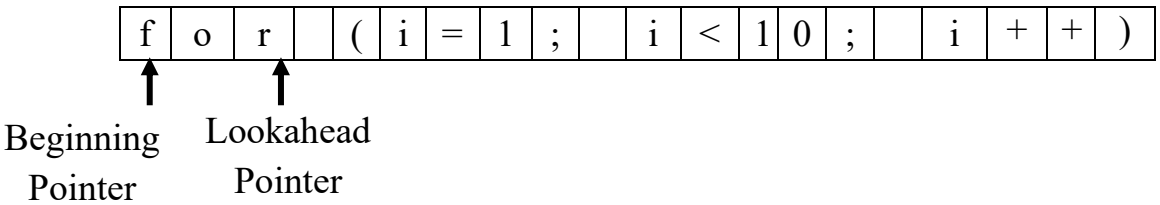
Note: These tables in above are saving in storage structure which called ***Symbol Table***.

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; i<=10; i++)** then the buffer will be

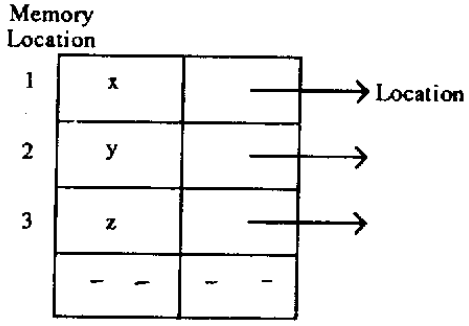


Symbol Tables

A symbol table is a set of locations containing a record for each identifier with fields for the attributes of the identifier. A symbol table allows us to find the record for each identifier (variable) and to store or retrieve data from that record quickly. Symbol table contains all information that must be passed between different phases of a compiler.

Gather information about names and constants which are in a program. For example, take an expression written in C such as: ***int x, y, z;***

The lexical analysis after going through this expression will enter x, y and z into the symbol table. This is shown in the figure given below.



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:

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: Token is a sequence of characters that can be treated as a single logical entity. In programming language, ***keywords***, ***constants***, ***identifiers***, ***strings***, ***numbers***, ***operators*** and ***punctuations*** symbols can be considered as tokens.

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"

For example, in C language, the variable declaration line
int value = 100; contains the tokens:

int (keyword)***, value*** (identifier)***, =*** (operator)***, 100 (constant) and ;*** (symbol).

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.

LEXICAL ERRORS:

Lexical errors are the errors thrown by your lexer when unable to continue. This means that there's no way to distinguish a lexeme as a valid token for your lexer. Simple ***panic-mode error handling*** system requires that we return to a high-level parsing function when a parsing or lexical error is detected. ***Panic mode recovery*** includes:

- Delete one character from the remaining input (the successive character). Example: printff---------- printf.
- Insert a missing character in to the remaining input.
  Ex: prite ------- printf
- Replace an incorrect character with the correct character.
- Ex: Printf ---------- printf

- Transpose two adjacent characters.
  Ex: rpintf ---------- printf

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".

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.

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language 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.

Special Symbols: A typical high-level language contains the following symbols:-

Arithmetic Symbols	**Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)**
Punctuation	**Comma(,), Semicolon(;), Dot(.), Arrow(->)**
Assignment	**=**
Special Assignment	**+=, /=, *=, -=**
Comparison	**==, !=, <, <=, >, >=**
Preprocessor	**#**
Location Specifier	**&**
Logical	**&, &&, |, ||, !**
Shift Operator	**>>, >>>, <<, <<<**

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*** ∪ ***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^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ 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

- Union: ***L*****U*****D*** = {a, b, c, 0,1}
- Concatenation: ***LD*** = {a0,a1, b0, b1, c0,c1}
- Exponentiation: L^2 = LL
- By definition: $L^0 = \{\in\}$

Regular Expressions

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as ***regular grammar***. The language defined by regular grammar is known as ***regular language***.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

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} ; where ∑ : indicates input set(i.e. input alphabet).

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, …..

Theorem1: the class of regular expressions is closed under union.

Theorem1: the class of regular expressions is closed under concatenation.

Regular Definitions

A regular definition gives names to certain regular expressions and uses those names in other regular expressions.
Representing occurrence of symbols using regular expressions:
letter = [a – z] or [A – Z]
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]
sign = [+ | -]

Representing language tokens using regular expressions:

Decimal = *sign*? *digit*+

Identifier = *letter (letter | digit)**

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 i.e. (**[0,9]**)

digits → digit^{+}

id → letter (letter | digit)*

number -->digit(.digit)$^{?}$**(e.[+ | -]**$^{?}$**digits)**$^{?}$

if → if

then → then

else → else

relop → < | > |<= |>= | == | < >

Notice: $R^{?}$Stand for zero or one occurrence of an event in R, that is, R+ Є .
(i.e. at most one occurrence of R).

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.

Design Regular Expressions –Examples

1- Design RE for the following languages over $\sum = (a, b)$:
   a) Language accepting strings of length exactly 2?
   b) Language accepting strings of at least 2?
   c) Language accepting strings of at most 2?

   Solution

   a) L1={aa, ab, ba, bb}; that is, RE= aa+ab+ba+bb
      RE= a(a+b)+b(a+b) = (a+b) (a+b)
   b) L2={aa, ab, ba, bb, aaa, …..} at least 2 means (min.=2, max.=up)
      RE= (a+b) (a+b) $(a+b)^*$
   c) L3= {Є, a, b, aa, ab, ba, bb} at most 2 means (min.=Є, max.=2)
      RE= Є+a+b+aa+ab+ba+bb= (Є +a+b)(ϵ +a+b)

2- Write RE for a language(L) containing string which end with 011 over $\sum = (0,1)$?
   Solution:
   $(0 + 1)^* = \{Є, 0, 1, 00, 11, 01, 10 \ldots\ldots\}$
   RE = $(0 + 1)^*$ 011

3- Write a RE for recognizing identifier?
   Solution:
   Letter= [A---Z, a----z] ; digit=[0---9]
   RE=letter $(\text{letter} + \text{digit})^*$
4- Write RE for L accepting all combination of a's and b's except null string?
   Solution: RE= $(a+b)^+$
5- Write RE for L accepting the string (S) which are starting with **a** and ending with **b** over $\sum = (0,1)$?
   Solution: RE= a $(a+b)^*$ b
6- Write a RE for L accepting the string 0's and 1's whose right end is 1 from 5 symbol?
   Solution: RE= $(0+1)^*$ 1 (0+1) (0+1) (0+1) (0+1)
7- Write the RE for L such that: L= { a^n b^m , (n+m) is even}
   Solution: n+m=even in two cases:

Case1: both n and m are even. Thus n,m =0,2,4,…

When n=0, m=0 ; $a^n b^m = a^0 b^0 = Є$

If n=2, m=2 ; $a^n b^m = a^2 b^2$ = aabb

RE= $(aa)^*$ $(bb)^*$

Case2: both n and m are odd (i.e. n=m=1,3,5,….)

When n=1 and m=1 thus n+m=1+1=2 is even. And a^1b^1 = ab.

If n=m=3; n+m=6; a^3b^3 = aaabbb; L={ab,aaabbb,aaaaabbbbb,..}

So, RE = $(aa)^*$ a $(bb)^*$ b

8- Write a RE for L containing strings of length two over $\sum = (a, b)$.

Solution: (a+b) thus means L= {a,b)

(a+b) (a+b) thus L= {aa,ab,ba,bb}

9- Write a RE for L containing even length strings over$\sum = (a, b)$.

10- Write a RE for L containing odd length strings over$\sum = (a, b)$.

11- Write a RE for L containing strings of length devisable by 3 over $\sum = (a, b)$.

12- Write a RE for L containing strings starting with the same symbol over$\sum = (a, b)$.

13- Write a RE for L containing strings starting with the different symbol over$\sum = (a, b)$.