**Case Study:**

**Presentation contents and mark split-up:**

|  |  |  |  |
| --- | --- | --- | --- |
| S.No | Content to be included | Marks split-up(10 Marks) | Viva(10 Marks) |
| 1 | Problem Statement | 2 |  |
| 2 | Language Description | 2 |
| 3 | Grammar | 3 |
| 4 | Finite Automata | 3 |

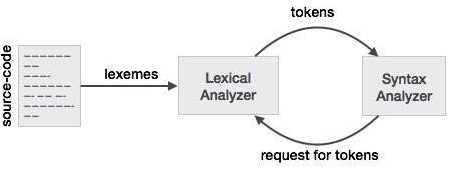
**Sample**

**Problem Definition:**

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

**Block Diagram:**



Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

int value = 100;

contains the tokens:

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

Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

**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.

**Strings**

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

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 | >>, >>>, <<, <<< |

### Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

## Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

**For example:**

int intvalue;

While scanning both lexemes till ‘int’, the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

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.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

## Operations

The various operations on languages are:

* Union of two languages L and M is written as

L U M = {s | s is in L or s is in M}

* Concatenation of two languages L and M is written as

LM = {st | s is in L and t is in M}

* The Kleene Closure of a language L is written as

L\* = Zero or more occurrence of language L.

## Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

* **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
* **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
* **Kleene closure** : (r)\* is a regular expression denoting (L(r))\*
* (r) is a regular expression denoting L(r)

## Precedence and Associativity

* \*, concatenation (.), and | (pipe sign) are left associative
* \* has the highest precedence
* Concatenation (.) has the second highest precedence.
* | (pipe sign) has the lowest precedence of all.

### Representing valid tokens of a language in regular expression

If x is a regular expression, then:

* x\* means zero or more occurrence of x.

i.e., it can generate { e, x, xx, xxx, xxxx, … }

* x+ means one or more occurrence of x.

i.e., it can generate { x, xx, xxx, xxxx … } or x.x\*

* x? means at most one occurrence of x

i.e., it can generate either {x} or {e}.

[a-z] is all lower-case alphabets of English language.

[A-Z] is all upper-case alphabets of English language.

[0-9] is all natural digits used in mathematics.

### 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 = [ + | - ]

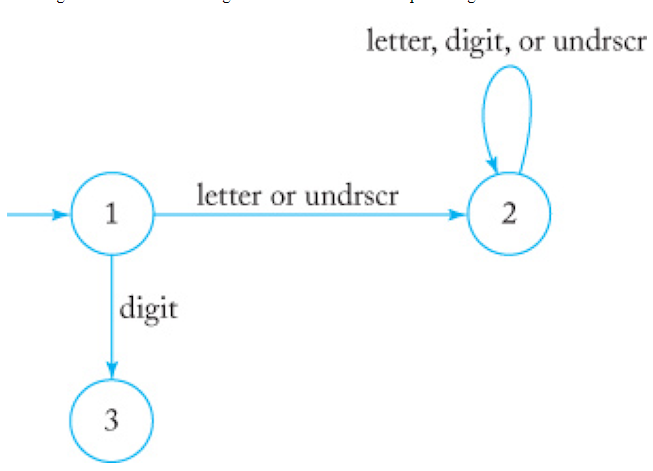
### Representing language tokens using regular expressions

Decimal = (sign)?(digit)+

Identifier = (letter)(letter | digit)\*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

**Automata for identifier:**



**Accepted strings:**

\_a

Avg\_

\_2c

**Unaccepted Strings:**

22

2c

2\_

$2

\*e

**Grammar:**

