LEXICAL ANALYSER Design

Brief Description about the project

In computer science, **lexical analysis** is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a **lexical analyzer**, **lexer**, or **scanner**. A lexer often exists as a single function which is called by a parser or another function.

Lexical analyzers are designed to recognize keywords , operators , and identifiers , as well as integers, floating point numbers , character strings , and other similar items that are written as part of the source program.

TOKEN

A **token** is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization**, and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parentheses as tokens, but does nothing to ensure that each "(" is matched with a ")".

Consider this expression in the C programming language:

```
sum=3+2;
```

Tokenized in the following table:

Lexeme	**Token type**
sum	Identifier
=	Assignment operator
3	Integer literal
+	Addition operator
2	Integer literal
;	End of statement

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing." If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures for general use, interpretation, or compiling.

PURPOSE OF THE PROJECT

The main purpose/goal of the project is to take in a c file, .c, and produce the sequence of tokens that can be used for the next stage in compilation.

SCOPE OF THE PROJECT

Lexical Analyzer will be a small product demonstrating the use of C language. It will assume the source program stores in a file which will be accessed using the file operations and also assuming that the input is taken from the output of the preprocessor. It is designed only for the C language, not for any other language. It will not be using any optimal data structures or algorithms in the initial versions.

This project will be designed only for the sub-set of C.

Sub- set of C :

1. It should identify all the keywords
2. It should identify identify all the Identifiers.
3. It should identify the literals, such as float , characters, string literals, decimals.
4. It should identify the arrays.

1. INTRODUCTION TO COMPILER

A compiler translates and/or compiles a program written in a suitable source language into an equivalent target language through a number of stages. Starting with recognition of token through target code generation provide a basis for communication interface between a user and a processor in significant amount of time.

A compiler is system software that converts a high-level programming language program into an equivalent low-level (machine) language program. It validates the input program conforming the source language specification and violation of the same is stipulated as error message or warnings. Obviously it attempts to mark and detail the mistakes done by the programmer . The idea is shown in Figure 1.

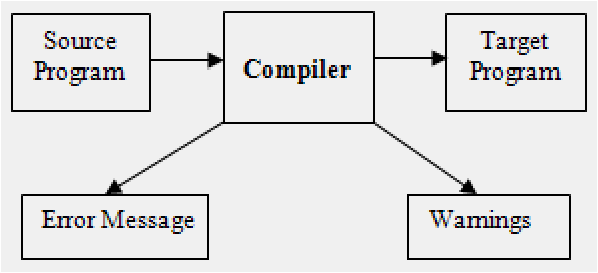


Figure 1: Working Methodology of a Compiler

Beginning with token recognition, it runs through generation of context free grammar, parsing sequence, checking acceptability, machine independence intermediate code generation to finally target code generation state. These act as a basis for communication interface between user and processor .

2. PHASES OF GENERAL COMPILER

Writing a compiler is a nontrivial task. It will be a very nice practice to structure its principles. Conceptually a compiler works in phases. The key phases include and undergo through Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Code Optimization, and Target Code Generation . These are shown in Figure 2.

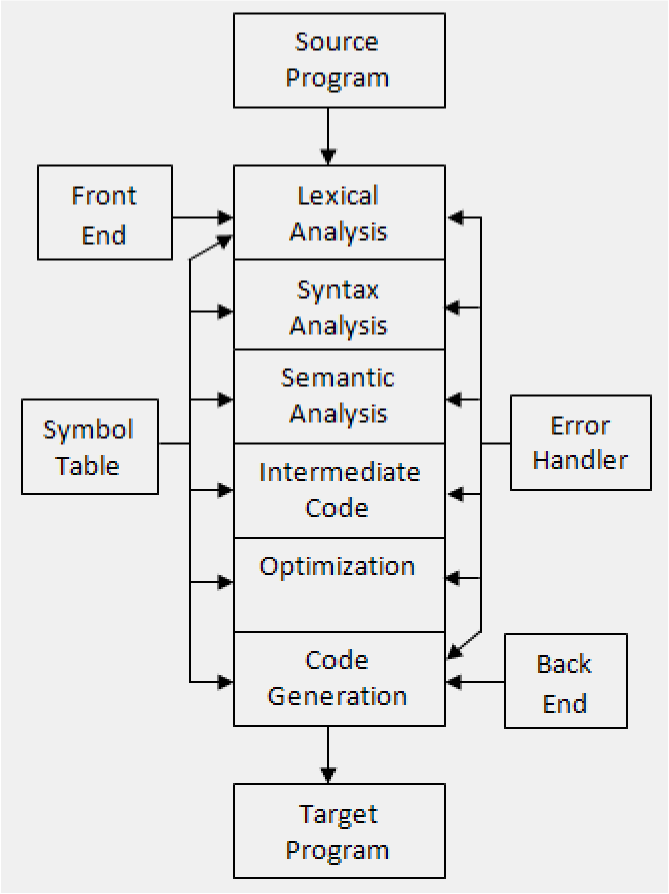
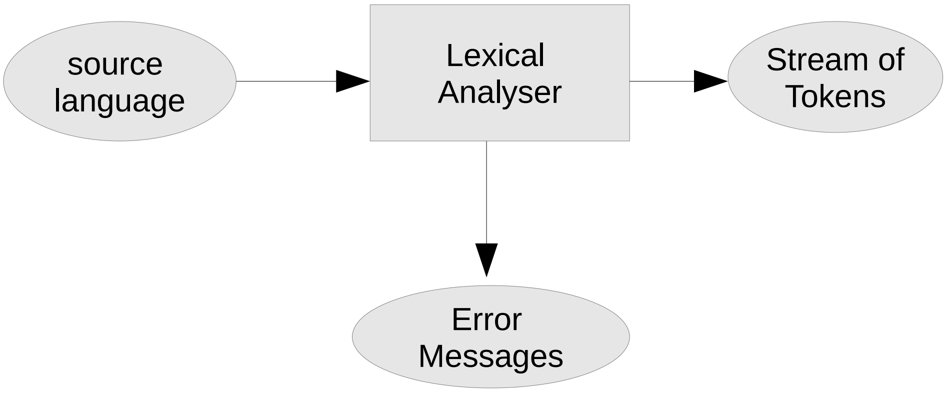


Figure 2: Phases of a Typical Compiler

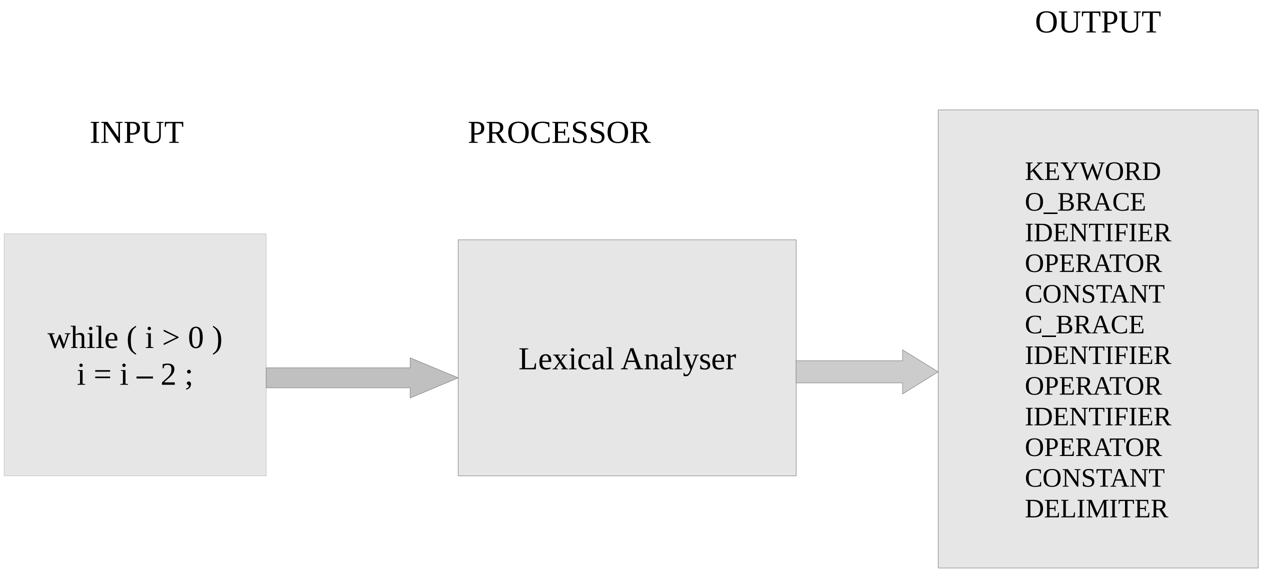
3. BASICS OF LEXICAL ANALYSIS

Lexical analysis or scanning is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. Tokens are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language: constants (integer, double, char, string, etc.), operators (arithmetic, relational, logical) and reserved words.

4. BLOCK DIAGRAM OF LEXICAL ANALYSER



For Example:



The lexical analyzer takes a source program as input, and produces a stream of tokens as output. The lexical analyzer might recognize particular instances of tokens such as:

3 or 255 for an integer constant token

"Fred" or "Wilma" for a string constant token

numTickets or queue for a variable token

Such specific instances are called lexemes. A lexeme is the actual character sequence forming a token, the token is the general class that a lexeme belongs to. Some tokens have exactly one lexeme (e.g., the > character); for others, there are many lexemes (e.g., integer constants).

The scanner is tasked with determining that the input stream can be divided into valid symbols in the source language, but has no smarts about which token should come where. Few errors can be detected at the lexical level alone because the scanner has a very localized view of the source program without any context. The scanner can report about characters that are not valid tokens (e.g., an illegal or unrecognized symbol) and a few other malformed entities (illegal characters within a string constant, unterminated comments, etc.) It does not look for or detect garbled sequences, tokens out of place, undeclared identifiers, misspelled keywords, mismatched types and the like. For 2 example, the following input will not generate any errors in the lexical analysis phase, because the scanner has no concept of the appropriate arrangement of tokens for a declaration. The syntax analyzer will catch this error later in the next phase.

```
int a double } switch b[2] =;
```

Furthermore, the scanner has no idea how tokens are grouped. In the above sequence, it returns b, [, 2, and] as four separate tokens, having no idea they collectively form an array access.

The lexical analyzer can be a convenient place to carry out some other chores like stripping out comments and white space between tokens and perhaps even some features like macros and conditional compilation (although often these are handled by some sort of preprocessor which filters the input before the compiler runs).

5. WORKING PRINCIPLE OF LEXICAL ANALYSER

A lexical analyzer (also known as lexer), a pattern recognition engine takes a string of individual letters as its input and divides it into tokens . Additionally, it also filters out whatever (usually white-space, newlines, comments, etc) separates the tokens. The main purpose of this phase is to make the subsequent phase easier .

Some key definitions related to this phase include:

Lex: A set of buffered input routines and constructs. It translates regular expression into lexical analyzer.

Tokens: Basic indivisible lexical unit or language elements. These are terminal symbols in a grammar, Constants, Operators, Punctuation, Keywords, Classes of sequences of characters with collective meaning, arbitrary integer values, etc that represent the lexemes .

Lexeme: These are original string (character sequence) comprised (matched) by an instance of the token. E.g. “sqrt” .

Lexical Analyzer is basically a part of compiler which:

i. Translates lexemes into tokens (arranged in symbol table for compilation references) with the help of Lex .

ii. Communicates with parser for serving token requests.

iii. skips over white spaces.

iv. Keeps track of current line number so that parser can detect errors .

Working methodology of lexical analyzer has been traced in some interesting phases as stated below:

First, it acts as an interface for parser and symbol table with input stream as reference as shown in

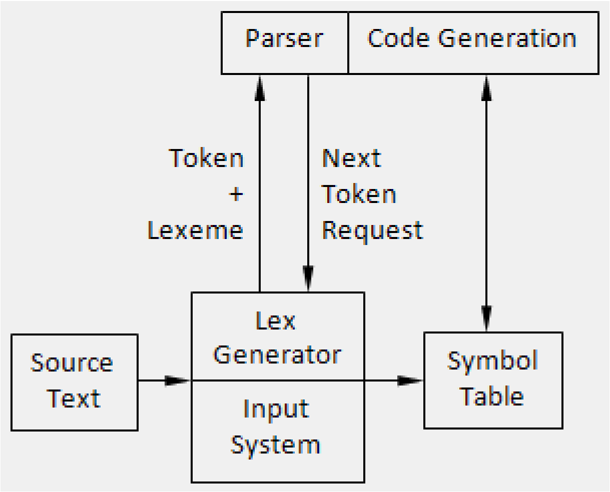


Figure 3: Lex as a Tokenizer

Second, internal working procedure of Lexical Analyzer that generates a stream of tokens. Suppose the pseudocode:

```
if( x * y < 10)
{
        Z = x;
}
```

Let’s consider the first statement of the above code. The corresponding token stream of pairs <type, value> is shown in Figure 4. Lex and input systems together constitute layers of Lexical Analyzer.

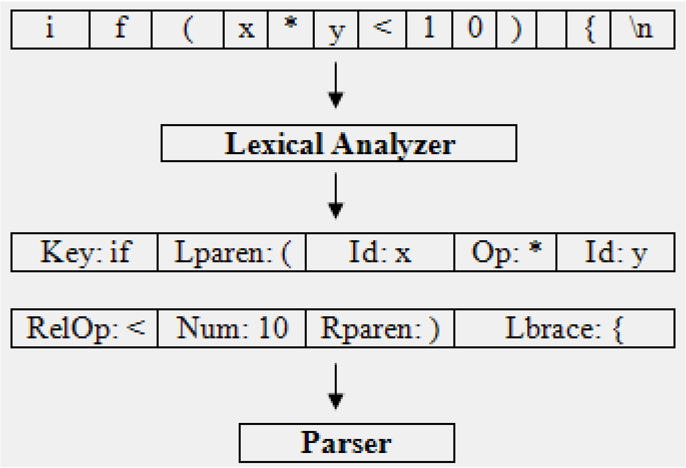


Figure 4: Output Stage of Lexical Analyzer

5.1 INPUT SYSTEM

Input System is the lowest-level layer of the lexical analyzer which consists of group of functions that actually read data from the operating system. This is the reason that refers the lexical analyzer as a distinct and independent module . This independency may derive several advantages such as:

1. Change in the phase doesn’t affect compiler as a whole.
2. Enhanced portability .
3. Efficient speed to read large data. Thus, optimized read time.

An input system must possess the following design criteria .

1. Efficient disk access.
2. Supported reasonable lexemes of finite length.
3. Faster routines as possible, with little or no copying of the input strings.

6. SAMPLE ALGORITHM FOR LEXICAL ANALYZER

Building a Lexical Analyzer needs a language that must describe the tokens, token codes, and token classification. It also needs to design a suitable algorithm to be implemented in program that can translate the language into a working lexical analyzer.

We have used C language in particular, for implementation as powerful tool enough to describe the symbols. The algorithm designed for the lexical analyser for the generation of tokens is named as Tokenizer and is written below.

```
ALGORITHM :  Tokenizer(S)
Where S = Input string.
Output :  A set of tokens

Step 1: Initialize S.
Step 2: Define symbol table.
Step 3: Repeat while scanning (left to right) S is not completed
        i. If blank (empty space)
                a. Neglect and eliminate it.

        ii. If operator op          // arithmetic, relational, etc.
                a. Find its type.
                b. Write op.

        iii. If keyword key         // if, while, for, etc.
                a. Write key.

        iv. If identifier id        // a, b, c, etc
                a. Write id.
```

```
    v. If special character sc        // (, ), etc.
        a. Write sc.
Step 4: Exit
```

7. DATA FLOW DIAGRAM

A data flow diagram (DFD) is a graphical representation of the "flow" of data through an information system. It differs from the flowchartas it shows the data flowinstead of the control flow of the program. A data flow diagram can also be used for the visualization of data processing(structured design).

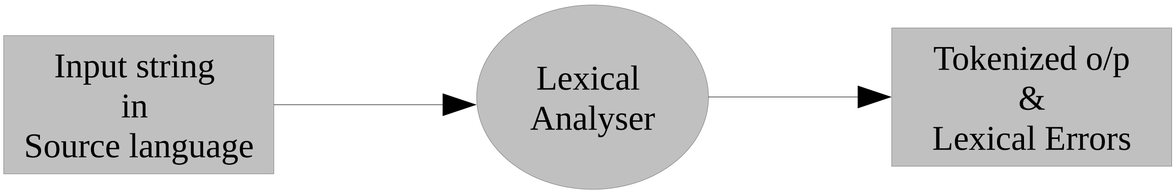


Fig : Zero level data flow diagram

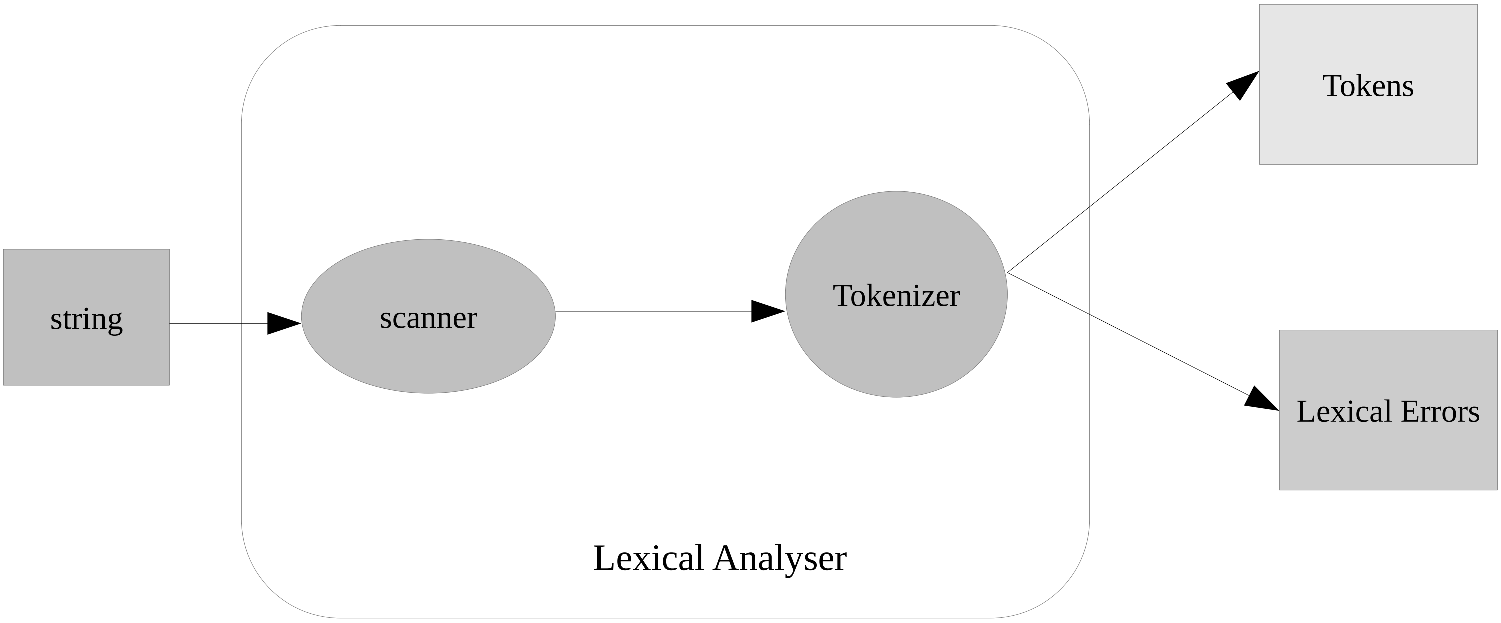


Fig : First level lexical data flow diagram

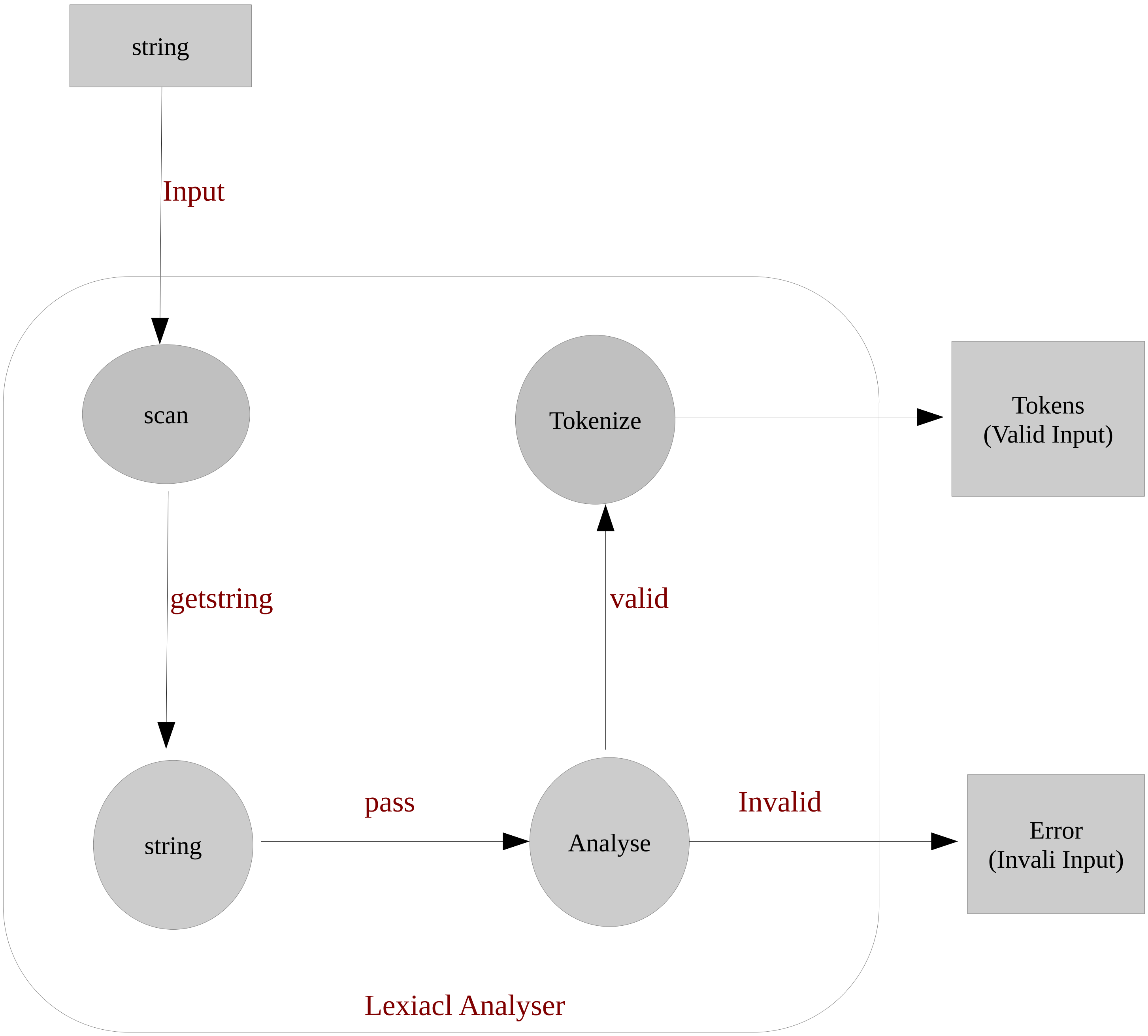
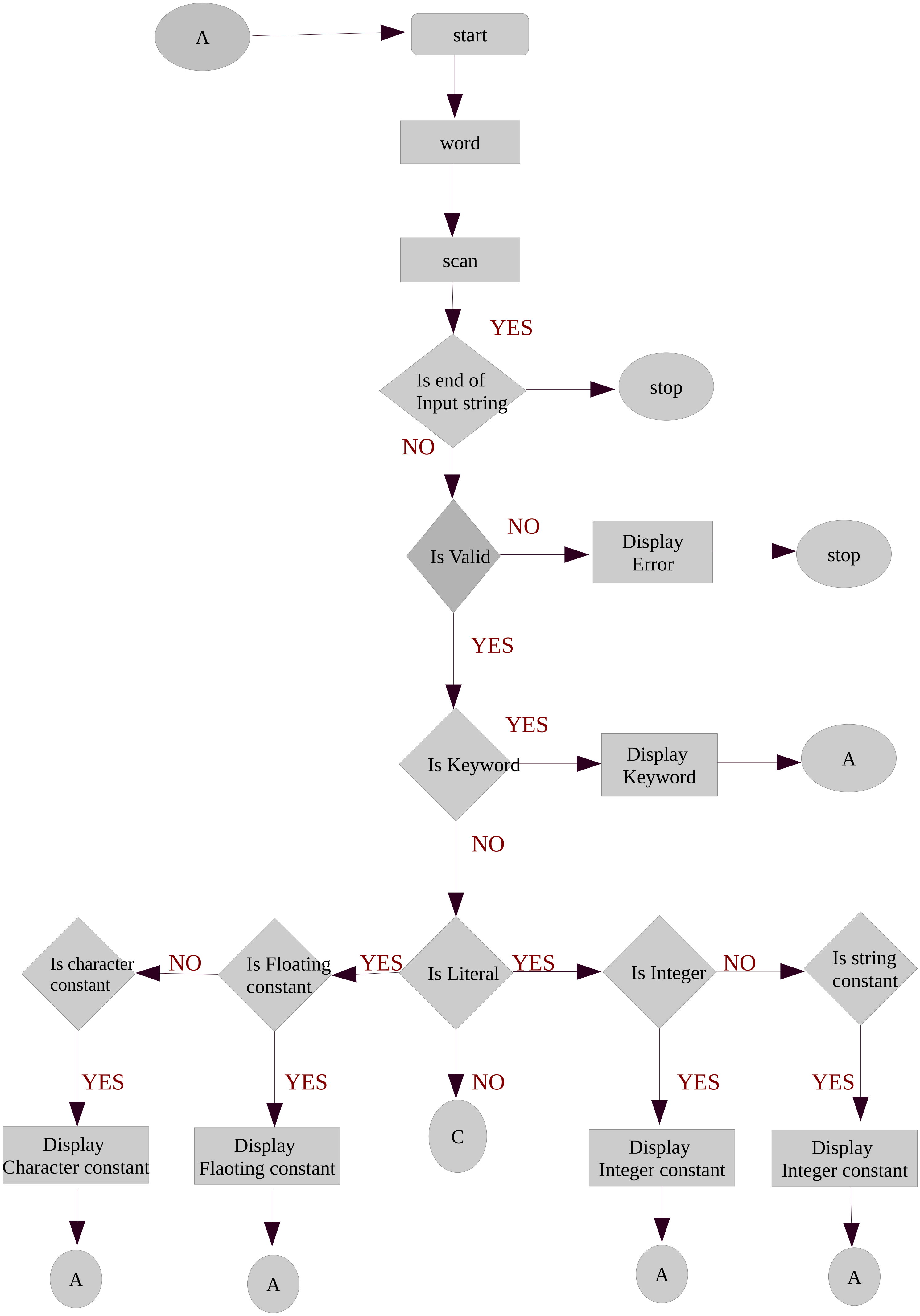
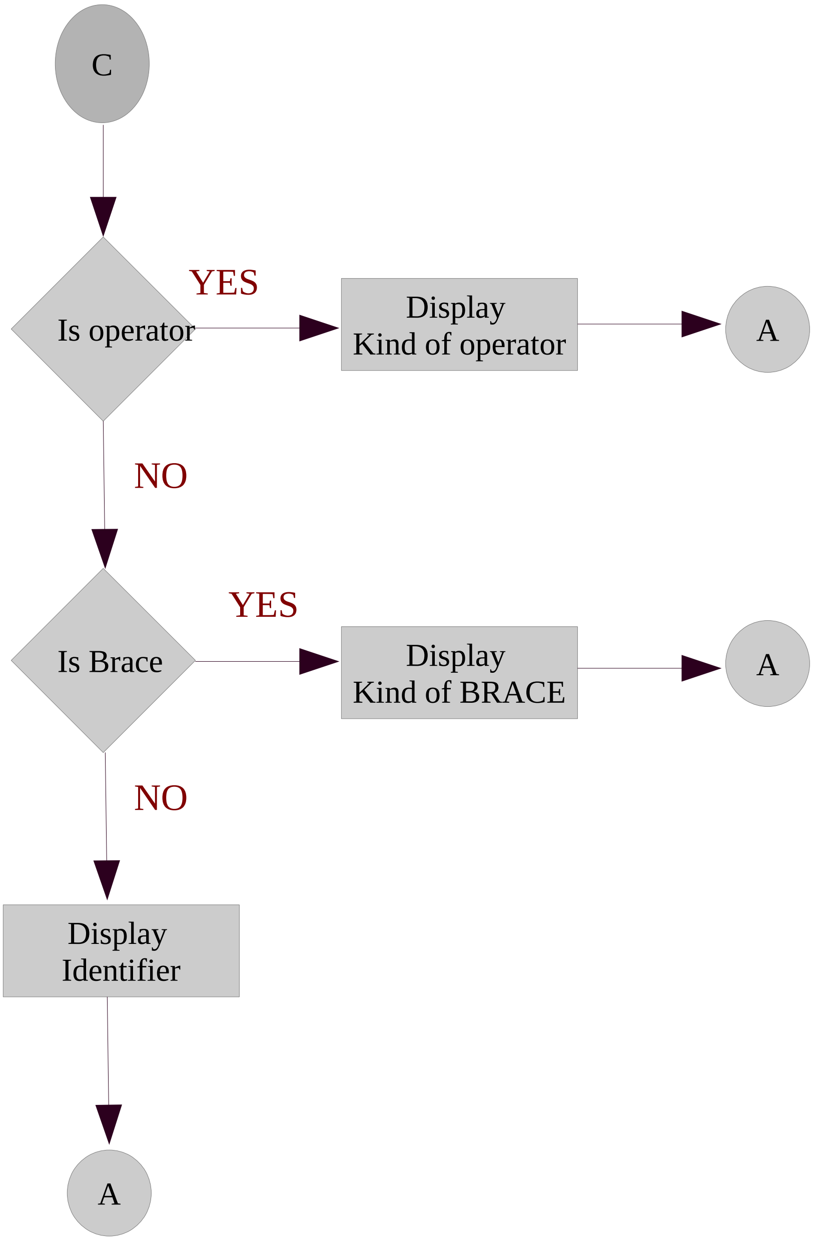


Fig : Second level data flow diagram

8. FLOW CHART


A
start
word
scan
YES
Is end of Input string
stop
NO
NO
Is Valid
Display Error
stop
YES
YES
Is Keyword
Display Keyword
A
NO
Is character constant
NO
Is Floating constant
YES
Is Literal
YES
Is Integer
NO
Is string constant
YES
YES
NO
YES
YES
Display Character constant
Display Flaoting constant
C
Display Integer constant
Display Integer constant
A
A
A
A


C
Is operator
YES
Display
Kind of operator
A
NO
Is Brace
YES
Display
Kind of BRACE
A
NO
Display
Identifier
A

9. EXPERIMENTAL RESULTS

Observation 1:

Valid input : for(x1 = 0; x1 <= 10; x1++);

Output Analysis:

```
for   :  Keyword
(     :  Special character
x1    :  Identifier
=     :  Assignment operator
0     :  Constant
;     :  Special character
x1     : Identifier
<=    : Relational operator
10    : Constant
;     : Special character
x1    : Identifier
+     : Operator
+     : Operator
)     : Special character
;     : Special character
```

Tokens generated.

Observation 2:

Invalid input: for(x1 = 0; x1 <= 19x ; x++);

Output Analysis:

```
for      : Keyword
(        : Special character
x1       :Identifier
```

=	: Assignment operator
0	: constant
;	: special character
x1	: Identifier
<=	: relational operator

Tokens cannot be generated.