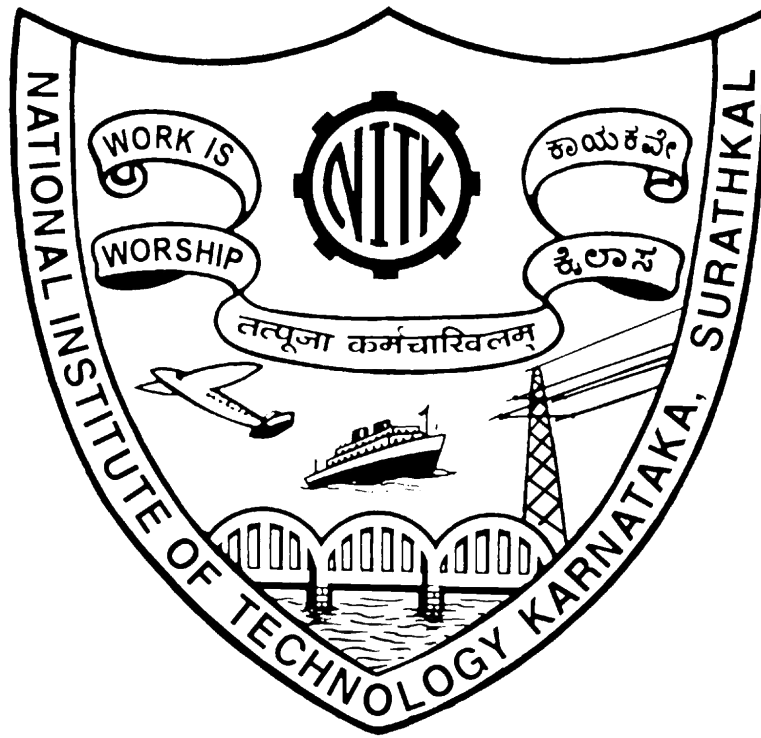


LEXICAL ANALYSER SCANNER FOR C LANGUAGE:

Compiler lab CO351



Akash S (13CO202)

Sudarshan S K (13CO146)

ABSTRACT

A Lexical analyzer processes the code and produces tokens. It is built using the lex/flex tools. This analyzer for the C language, supports nested comments and returns meaningful errors if there are any. This program also takes care of unmatched parentheses and braces.

CONTENTS

- Introduction
 - 1. Lexical analyzer
 - 2. Flex script
 - 3. C program
- Design of programs
- Test cases
 - 1. Without error
 - 2. with errors
- Implementation
- Results / future work
- Reference

List of figures / tables

- Table of Test cases for parser
- Screenshots of result

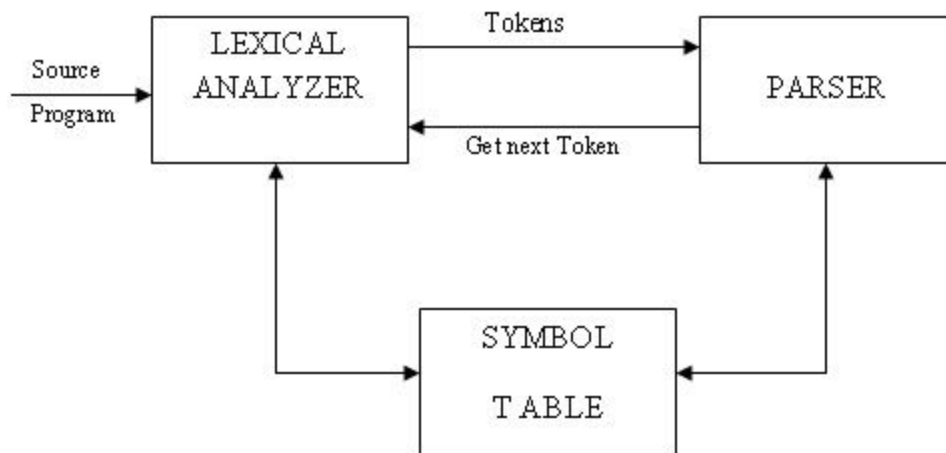
INTRODUCTION

The scanner performs lexical analysis of a certain program (in our case, the Simple c program). It reads the source program as a sequence of characters and recognizes "larger" textual units called tokens. our scanner following these lexical rules:

- Case insensitive
- All English letters (upper and lower), digits, and extra characters as seen below, plus whitespace.
- Identifiers
 - begin with a letter, and continue with any number of letters
 - assume no identifier is longer than 8 characters
- Keywords (reserved), include: start finish then if repeat var int float do read print void return dummy program
- Relational operators, include: == < > != => =<
- Other operators, include: = : + - * / %
- Delimiters, include: . () , { } ; []
- Numbers:
 - any sequence of decimal digits, no sign
 - assume no number is longer than 8 digits
- Other assumption:
 - No white space needed to separate tokens except when this changes the tokens (as x y vs xy)
- Comments start with // and end with newline

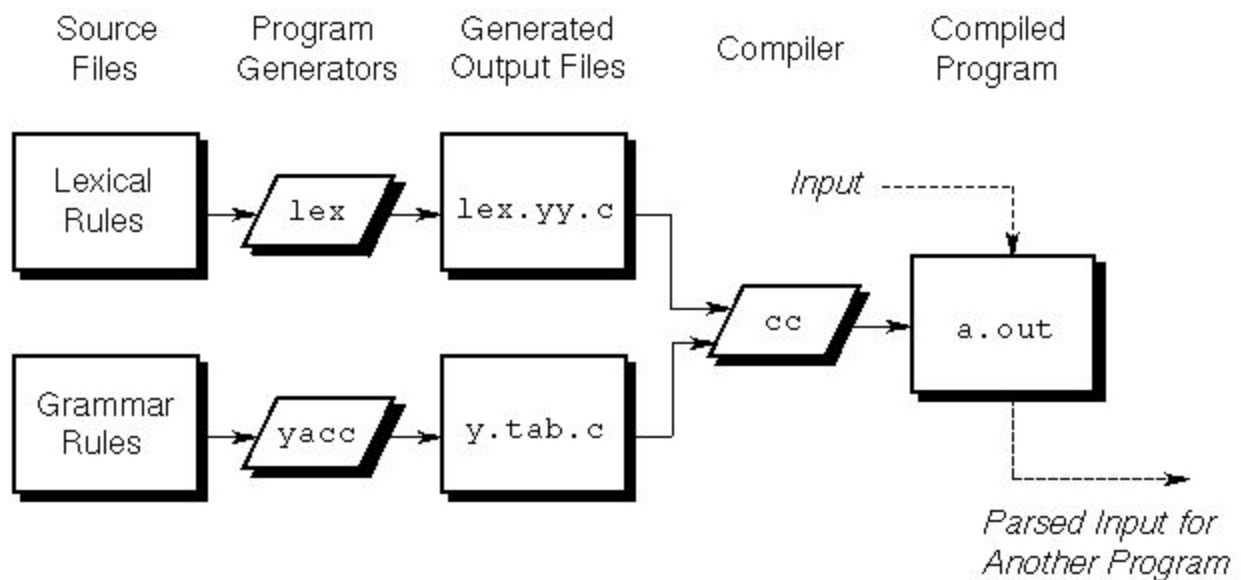
Lexical analyzer

Lexical analyzer (or scanner) is a program to recognize tokens (also called symbols) from an input source file (or source code). Each token is a meaningful character string, such as a number, an operator, or an identifier. In computer science, lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program that performs lexical analysis may be called a lexer, tokenizer or scanner



Flex script

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. In stead of writing a scanner from scratch, you only need to identify the vocabulary of a certain language (e.g. Simple), write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a scanner for you. FLEX is generally used in the manner depicted here:



ZK-0455U-R

First, FLEX reads a specification of a scanner either from an input file *.lex, or from standard input, and it generates as output a C source file lex.yy.c. Then, lex.yy.c is compiled and linked with the "-lfl" library to produce an executable a.out. Finally, a.out analyzes its input stream and transforms it into a sequence of tokens.

- ***.lex** is in the form of pairs of regular expressions and C code.
- **lex.yy.c** defines a routine yylex() that uses the specification to recognize tokens.
- **a.out** is actually the scanner!

C program

Here our c program is input as test cases of lexical analyzer so that it can generate output file containing all the relevant information and analysis about that test cases(c program), and meaningful errors if there are any.

```
#include<stdio.h>
/* multi line comment begins here
/* this may be incorrect*/
ends here*/
int main()
{
int count, n, a=0, b=1,display;
printf("Enter number of terms: ");
scanf("%d",&n);
printf("Fibonacci Series: %d+%d+", a, b); /* Displaying first two terms */
count=2; /* count=2 because first two terms are already displayed. */
while (count<n)
{
display=a+b;
a=b;
b=display;
++count;
printf("%d+",display);
}
return 0;
}
```

Design of programs

code - LA.l

```
%{
#include<stdio.h>
int c=0;
int f=0;
int cm=0;
%}
h #include<[a-z]+\h>
key
(auto) | (break) | (case) | (char) | (const) | (continue) | (default) | (do) | (double) | (else) | (enum) | (extern) | (
float) | (for) | (goto) | (if) | (int) | (long) | (register) | (return) | (short) | (signed) | (sizeof) | (static) | (struct) | (
switch) | (typedef) | (union) | (unsigned) | (void) | (volatile) | (while)
co "("
fo "{"
cc ")"
fc "}"
id [a-z]+[0-9]*_?
conint ([1-9][0-9]*(\.[0-9]+)? | (0)
constr \"[^\"]\"
op [\-\+\=\*\^\^]
rop < | > | "==" | "<=" | "*"=" | "/"=" | ">="
func [a-z0-9]+([^\^]*)
slcomm "//".*
mlcomm "/*"([^\^] | (\^[^/]))* \^+ \^
new \n
sc ; | ,
ml "/*" | "*/"
%%

{h}    {printf("%s is a header\n",yytext);}

{key}   {printf("%s is a keyword\n",yytext);}

{id}    {printf("%s is an identifier\n",yytext);}

{conint} {printf("%s is an integer constant\n",yytext);}

{constr} {printf("%s is a string\n",yytext);}

{op}    {printf("%s is a operator\n",yytext);}

{rop}   {printf("%s is a relational operator\n",yytext);}

{func}  {printf("%s is a function call\n",yytext);}

{co}    {c++;}
```



```

{cc}    {c-;}
{fo}    {f++;}
{fc}    {f-;}

<<EOF>> {
        if(c!=0)
        {
            printf("Unmatched parentheses\n");
        }
        if(f!=0)
        {
            printf("Unmatched braces  %d \n",f);
        }
        return 0;
    }

{new}    {if(c!=0)
        {
            printf("Unmatched parentheses.\n");
        }}

{slcomm}    {printf("%s is a single-line comment\n",yytext);}

{mlcomm}    {printf("%s is a multi-line comment\n",yytext);}

{ml}        {printf("Error: Comment not closed.\n");}

{sc}
[ ^ ]        {printf("Unknown symbol %s \n", yytext);}

%%
int main(int argc, char **argv)
{
    FILE *in;
    if(argc==2 && (in=fopen(argv[1],"r")))
    {
        yyin=in;
    }
    yylex();
    return 0;
}

```

TEST CASES TABLE

Serial Number	Test Case	Expected Output	Status
1	int a=10;	Keyword: int Identifier: a Operator: = Constant: 10	Working
2	<u>printf</u> ("Hello World");	Function call <u>printf</u>	Working
3	b = a+10	Identifier: b, a Operator: =, + Constant: 10 Error missing semicolon	Working
4	#include< <u>stdio.h</u> >	Header file	Working
5	//this is a single line comment	Single line comment	Working
6	/*this is not A Single line comment*/	Multiple line comment	Working
7	/*this is supposed /* To be a nested /* Comment */	Error – Unmatched comment delimiter	Working
8	<u>printf</u> ("unclosed);	Function call – <u>printf</u> Error – Unmatched quotes	Working
9	void function({}	Keyword: void Error – Unmatched parenthesis	Working
10	int main(){ b = f?3; return 0;	Keyword: int, main, return Function call – main Identifier: b,f Constant: 3, 0 Error - Unknown symbol ? Error – Unmatched braces	Working

IMPLEMENTATION

keyword definitions -

```
h #include<[a-z]+\.\h>
```

```
key
```

```
(auto)|(break)|(case)|(char)|(const)|(continue)|(default)|(do)|(double)|(else)|(enum)|(extern)|(float)|(for)|(goto)|(if)|(int)|(long)|(register)|(return)|(short)|(signed)|(sizeof)|(static)|(struct)|(switch)|(typedef)|(union)|(unsigned)|(void)|(volatile)|(while)
```

To take care of parentheses -

```
{co}    {c++;}
```

```
{cc}    {c--;}
```

```
{fo}    {f++;}
```

```
{fc}    {f--;}
```

```
<<EOF>> {
        if(c!=0)
        {
            printf("Unmatched parentheses\n");
        }
        if(f!=0)
        {
            printf("Unmatched braces  %d \n",f);
        }
        return 0;
    }
```

```
{new}    {if(c!=0)
        {
            printf("Unmatched parentheses.\n");
        }}
}
```















For comments -

```
{slcomm}    {printf("%s is a single-line comment\n",yytext);}
```

```
{mlcomm}    {printf("%s is a multi-line comment\n",yytext);}
```

```
{ml}        {printf("Error: Comment not closed.\n");}
```

```

akash@Boo: /media/akash/WORKSPACE/Course Work/Compilers
 akash@Boo:/media/akash/WORKSPACE/Course Work/Compilers$ ./a.out sample.c
#include<stdio.h> is a header
/* multi line comment begins here
 /* this may be incorrect*/ is a multi-line comment
ends is an identifier
here is an identifier
Error: Comment not closed.
int is a keyword
main() is a function call
int is a keyword
count is an identifier
 n is an identifier
a is an identifier
= is a operator
 0 is an integer constant
b is an identifier
= is a operator
 1 is an integer constant
display is an identifier
Unknown symbol ♦
 Unknown symbol ♦
printf("Enter number of terms: ") is a function call
Unknown symbol ♦
 Unknown symbol ♦
scanf("%d",&n) is a function call
Unknown symbol ♦
 Unknown symbol ♦
printf("Fibonacci Series: %d+%d+", a, b) is a function call
Unknown symbol ♦
 Unknown symbol ♦
/* Displaying first two terms */ is a multi-line comment
count is an identifier
= is a operator
 2 is an integer constant
Unknown symbol ♦
 Unknown symbol ♦
/* count=2 because first two terms are already displayed. */ is a multi-line comment
while is a keyword
count is an identifier
< is a relational operator
n is an identifier
display is an identifier
= is a operator
 a is an identifier
+ is a operator
 b is an identifier
Unknown symbol ♦
Unknown symbol ♦
a is an identifier
= is a operator
b is an identifier
Unknown symbol ♦
Unknown symbol ♦
b is an identifier
= is a operator
display is an identifier
Unknown symbol ♦
 Unknown symbol ♦

```

Results / future work

Result of this project will be in form of a lex file which is an c language scanner and analyzer. which reads the source program as a sequence of characters and recognizes "larger" textual units called tokens as mentioned in introduction section. Future work of this project will be to develop one full fledged scanner with higher level of functionality and detection ability to improve the current work. Upcoming work will be to develop a parser using yacc/bison, a semantic checker and intermediate code generator for c language and make sure that all the component of project can work with each other to implement and small scale compiler.

References

- [1] <http://alumni.cs.ucr.edu/~lgao/teaching/>
- [2] <http://www.codeproject.com/Articles/>
- [3] <https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html>