Lexical Analyser for C

Aishwarya Rajan - 13CO109
Anagh Singh - 13CO110

January 28, 2016

Department of Computer Science and Engineering, NITK

Contents

List of Figures

Listings

Abstract

Lexical analyzer for c language is developed using Lex/Flex tools available that analyse C programs. This analyser supports nested comments and returns meaningful errors if there are any. This program also analyses all the comments and strings that it gets as input till the end of the file.

1 Introduction

Lexical analysis is the process of analysing a stream of individual characters (normally arranged as lines), into a sequence of lexical tokens (tokenization - for instance of "words" and punctuation symbols that make up source code) to feed into the parser. Roughly the equivalent of splitting ordinary text written in a natural language (e.g. English) into a sequence of words and punctuation symbols. Lexical analysis is often done with tools such as lex, flex and jflex.

Strictly speaking, tokenization may be handled by the parser. The reason why we tend to bother with tokenising in practice is that it makes the parser simpler, and decouples it from the character encoding used for the source code.

1.1 Lexical Analysis

1.1.1 What is a token?

In computing, a token is a categorized block of text, usually consisting of indivisible characters known as lexemes. A lexical analyzer initially reads in lexemes and categorizes them according to function, giving them meaning. This assignment of meaning is known as tokenization. A token can look like anything: English, gibberish symbols, anything; it just needs to be a useful part of the structured text. Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer such as lex. The lexical analyzer reads in a stream of lexemes 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.

1.1.2 What is a pattern?

There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token. Regular expressions are an important notation for specifying patterns. For example, the pattern for the Pascal identifier token, id, is:

id →letter (letter | digit)*

1.1.3 Finite State Automaton

An FSA is usually used to do lexical analysis.

An FSA consists of states, starting state, accept state and transition table. The automaton reads an input symbol and moves the state accordingly. If the FSA reaches the accept state after the input string is read until its end, the string is said to be accepted or recognized. A set of recognized strings is said to be a language recognized by the FSA.

1.1.4 Regular Expressions

The regular expressions over an alphabet specify a language according to the following rules. ϵ is a regular expression that denotes , that is, the set containing the empty string. If a is a symbol in alphabet, then a is a regular expression that denotes a, that is, the set containing the string a. Suppose r and s are regular expression denoting the languages $L(r)$ and $L(s)$. Then

- $(r)|(s)$ is a regular expression denoting $L(r)UL(s)$.
- $(r)(s)$ is a regular expression denoting $L(r)L(s)$.
- $(r)*$ is a regular expression denoting $(L(r))*$.
- $(r)|(s)$ is a regular expression denoting $L(r)UL(s)$.

1.2 Flex Script

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. In stead of writing a scanner from scratch, we 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 it.

$flex$ is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, $lex.yy.c$ by default, which defines a routine $yylex()$. This file can be compiled and linked with the flex runtime library to produce an executable.

When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

The internals of a Flex script are as follows: These programs perform character parsing and tokenizing via the use of a deterministic finite automaton (DFA). A DFA is a theoretical machine accepting regular languages. These machines are a subset of the collection of Turing machines. DFAs are equivalent to read-only right moving Turing machines. The syntax is based on the use of regular expressions

The flex input file consists of three sections, separated by a line containing only '%%'.

```
definitions
%%
rules
%%
user code
```

Character classes are expanded immediately when seen in the flex input. This means the character classes are sensitive to the locale in which flex is executed, and the resulting scanner will not be sensitive to the runtime locale. This may or may not be desirable.

1.2.1 Input Matching

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined, the text corresponding to the match (called the token) is made available in the global character pointer yytext, and its length in the global integer yyleng. The action corresponding to the matched pattern is then executed, and then the remaining input is scanned for another match.

If no match is found, then the default rule is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest valid flex input is:

```
%%
```

1.2.2 Generated Scanner

The output of flex is the file lex.yy.c, which contains the scanning routine yylex(), a number of tables used by it for matching tokens, and a number of auxiliary routines and macros. By default, yylex() is declared as follows:

```
int yylex()
    {
    ... various definitions and the actions in here ...
    }
```

Whenever yylex() is called, it scans tokens from the global input file yyin (which defaults to stdin). It continues until it either reaches an end-of-file (at which point it returns the value 0) or one of its actions executes a return statement. If yylex() stops scanning due to executing a return statement in one of the actions, the scanner may then be called again and it will resume scanning where it left off. The scanner writes its ECHO output to the yyout global (default, stdout), which may be redefined by the user simply by assigning it to some other FILE pointer.

1.3 C Program

This section contains the C progam which we provide as input to our scanner.

```
/* C Program to Swap two numbers
 **** Using temporary variable
*/
#include <stdio.h>

int main()
{
   int x, y, temp;

   printf("Enter the value of x and y\n");
   scanf("%d%d", &x, &y);
   x=2;
   printf("Before Swapping\nx = %d\ny = %d\n",x,y);

   temp = x;
   x    = y;
   y    = temp;

   printf("After Swapping\nx = %d\ny = %d\n",x,y);

   return 0;
}
```

This is another program that we provide as input:

```c
/* C Program to Swap two numbers
 */Using temporary variable*/
*/
#include <stdio.h>

int main()
{
   int x, y, temp;

   printf("Enter the value of x and y\n");
   scanf("%d%d", &x, &y);

   printf("Before Swapping\nx = %d\ny = %d\n",x,y);

   temp = x;
   x    = y;
   y    = temp;

   printf("After Swapping\nx = %d\ny = %d\n",x,y);

   return 0;
}
```

2 Design of Program

2.1 Code

This section contains our actual scanner. The code for it is:

```c
%{
#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#include <string.h>

int comment=0,bracCount=0;

char* ident[50]; //Symbol Table
char* constant[50]; //Constants Table
int it=0, ct=0;

#define OPERATOR    1
#define LCURLY      2
#define RCURLY      3
#define COMMENT     4
#define SEMICOLON   5
#define KEYWORD     6
#define IDENTIFIER  7
#define INTEGER     8
#define STRING      9
#define FUNC        10
#define FLOAT       11
#define PUNCTUATOR  12

```

```
%}

/* Definitions for finite Automata */

digit [0-9]

alpha [a-zA-Z_]

comment (\/\/.*)

comstart (\/\*)

comend (\*\/)

keyword "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"

relop >|<|<=|>=|!=

delimiter [ \t\v\n]+

%%



^#([-a-zA-Z0-9.]|{relop}|{delimiter})* {
                        fprintf(yyout,"\n%s\t\t\tPREPROCESSOR DIRECTIVE \t\t\t\t",yytext);

                        }

{comment}               {
                if(comment<=0) fprintf(yyout,"\n//\t\t\t\t\t LINE COMMENT \t\t\t\t\t\t");

                }

{comstart}                {

                          comment++;

                          if(comment>1)

                            fprintf(yyout,"\nERROR :: Nested Comment found.");

                          else

                            fprintf(yyout,"\n/*\t\t MultiLine Comment Begins\t\t\t");

                        }


{comend}                {

                          if(comment>0)

                           {
```

```
                        comment--;

                        fprintf(yyout,"\n*/\t\tMultiLine Comment Ends\t\t\t");

                    }

                    else

                        fprintf(yyout,"\n\tERROR: */ occurs before : /* ");

                }

{keyword}                   {if(comment<=0) fprintf(yyout,"\n%s\t\tKEYWORD\t\t",yytext);}

{alpha}+\(                  {if(comment<=0) fprintf(yyout,"\n%s\t \tFUNCTION\t\t",yytext);}

{alpha}({alpha}|{digit})*   {
                if(comment<=0)
                {
                 fprintf(yyout,"\n%s\t\tIDENTIFIER\t\t ",yytext);
                 ident[it++]=yytext;

                }
            }

{digit}+                    {
               if(comment<=0) {
                 fprintf(yyout,"\n%s\t\tNUMERIC CONSTANT\t\t ",yytext);
                 constant[ct++]=yytext;
               }
            }

{digit}+\.{digit}+          {if(comment<=0) {fprintf(yyout,"\n%s \t\t FLOAT CONSTANT\t\t ",yytext);
                 constant[ct++]=yytext;
            }}

\"(\\.|[^\\"])*\"           {if(comment<=0) {fprintf(yyout,"\n%-39s STRING LITERAL\t\t ",yytext);
                 //constant[ct++]=yytext;
            }}

\"(\\.|[^\\"])*             {if(comment<=0) printf("ERROR: End of string not found.\n");}

"{"                     {if(comment<=0) {fprintf(yyout,"\n%s\t\tOPEN BRACKET\t\t ",yytext);bracCount++;}}

"}"                     {if(comment<=0) {fprintf(yyout,"\n%s\t\tCLOSE BRACKET\t\t ",yytext);bracCount--;}}

","|";"|":"|"("|")"|"."     {if(comment<=0) fprintf(yyout,"\n%s \t\tPUNCTUATOR\t\t ",yytext);}

```

```
"&"|"!"|"~"|"+"|"-"|"*" {if(comment<=0) fprintf(yyout,"\n%s \t\
    tOPERATOR\t\t ",yytext);}
"/"|"%"|"<"|">"|"^"|"|" {if(comment<=0) fprintf(yyout,"\n%s \t\
    tOPERATOR\t\t ",yytext);}
"?"|"="|{relop}|"..."   {if(comment<=0) fprintf(yyout,"\n%s \t\
    tOPERATOR\t\t ",yytext);}
">>="|"<<="|"+="|"-="   {if(comment<=0) fprintf(yyout,"\n%s \t\
    tOPERATOR\t\t ",yytext);}
"*="|"/="|"%="|"&="     {if(comment<=0) fprintf(yyout,"\n%s \t\
    tOPERATOR\t\t ",yytext);}
"^="|"|="|">>"|"<<"|"+" {if(comment<=0) fprintf(yyout,"\n%s \t\
    tOPERATOR\t\t ",yytext);}
"--"|"->"|"&&"|"||"|"<=" {if(comment<=0) fprintf(yyout,"\n%s \t\
    tOPERATOR\t\t ",yytext);}

{delimiter} {;}

%%

/* MAIN PROGRAM */

int main(int argc,char **argv){

if(argv[1]==NULL)

 printf("Error opening file. Usage ./a.out <filename>\n");

else{

    yyin=fopen(argv[1],"r");

    printf("\n\t\t TOKEN IDENTIFICATION:\n Lexeme\t\t\tToken\n");

    yylex();

    if(comment!=0)

      printf("\nERROR: Comment does not end\n");

    if(bracCount!=0)

      printf("\nERROR: Bracket mismatch\n");

    printf("\n");

   printf("\n it:%d, ct:%d\n*************Symbol Table*************\
       n",it,ct);
   int index,k,j,flag;
   char x[50][50];
    for(index=0;index<it;index++){
        for(k=0;k<strlen(ident[index]);k++)
        {
            if (!isalnum(*(ident[index]+k))){
                x[index][k]='\0';
                break; }
            x[index][k]=*(ident[index]+k);
        }
        flag=0;
        for(j=0;j<index;j++)
            if(strcmp(x[j],x[index])==0) {flag=1; break;}

        if(!flag)
```

```
            printf("—>%s\n",x[index]);}
     printf("\n************************************\n");

     printf("\n************Constants Table*********\n");
      for(index=0;index<ct;index++){

     for(k=0;k<strlen(constant[index]);k++)
            {
                 if ((*(constant[index]+k))==';'){
                      x[index][k]='\0';
                      break; }
                 x[index][k]=*(constant[index]+k);
            }
            printf("—>%s\n",x[index]);
}
     printf("\n************************************\n");

   }

}

int yywrap()
{
return(1);
}
```

2.2 Explanation

Given lex.y file is code for our lexical analyser which is takes a C program as input and scan and analyses that code and works as given below.

2.2.1 Alphabet keywords

They are already predefined in their respective section.

2.2.2 To keep track of code blocks

This will take a section which is between/before or after comments (excludes comments).

2.2.3 To resolve nested comments

We resolved this situtation by keeping integer track of the depth of comments.

2.2.4 String errors

These are solved by checking last character of given string to check if it has ended or not.

3 Test Cases

S.No	Test case	Steps	Expected Results	Status
1	Identifier	Enter an identifier Enter an identifier Enter an identifier Enter an identifier	Var1 VALID IDENTIFIER 1dr INVALID IDENTIFIER _var VALID IDENTIFIER V1:b INVALID IDENTIFIER	Working
2	Integer Constant(int)	Enter an integer Enter an integer Enter an integer Enter an integer Enter an integer Enter an integer	3 VALID INTEGER 2.0 INVALID INTEGER 23,333 INVALID INTEGER -55 VALID INTEGER +10 VALID INTEGER 12938365675 INVALID INTEGER 09 INVALID INTEGER	Working
3	String Constant	Enter a string Enter a string Enter a string Enter a string	“Horse” VALID STRING “”Horse” INVALID STRING Horse INVALID STRING “100, Rs” VALID STRING	Working
4	Character Constant	Enter a character Enter a character Enter a character Enter a character Enter a character Enter a character Enter a character	’A’ VALID CHARACTER “n” INVALID CHARACTER ’kg’ INVALID CHARACTER ’’⁀VALID CHARACTER ’V́ALID CHARACTER ’iN’ INVALID CHARACTER ’9’ VALID CHARACTER	Working
5	Relational Operators	Enter an R-operator Enter an R-operator Enter an R-operator Enter an R-operator Enter an R-operator Enter an R-operator	>VALID R-OPERATOR >= VALID R-OPERATOR = INVALID R-OPERATOR == VALID R-OPERATOR != VALID R-OPERATOR - INVALID R-OPERATOR	Working
6	Real Constant	Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant	5.6 VALID CONSTANT 3.4.5 INVALID CONSTANT 34 INVALID CONSTANT 34 456 INVALID CONSTANT 34,477 INVALID CONSTANT -364.8 VALID CONSTANT +ie INVALID CONSTANT +3.0 VALID CONSTANT 314159E-5L VALID CONSTANT	Working

7	Keywords	Enter a keyword Enter a keyword Enter a keyword Enter a keyword Enter a keyword Enter a keyword Enter a keyword	double VALID KEYWORD main INVALID KEYWORD case VALID KEYWORD INT INVALID KEYWORD while VALID KEYWORD yes! INVALID KEYWORD auto VALID KEYWORD	Working
8	Nested Comments	Enter a comment Enter a comment Enter a comment	/*hello /*world*/ VALID /*hi /*hello*/*/ INVALID /*yes*/*/ INVALID	
9	Delimiters	Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter	{ VALID ; VALID : VALID , VALID # VALID [VALID) VALID INVALID . INVALID	Working
10	Unary Operators	Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator	++ VALID + VALID sizeof VALID * VALID & VALID VALID ! VALID / INVALID . INVALID	Working
11	Binary Operators	Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator	+ VALID - VALID / VALID * VALID —— VALID ++ INVALID ! INVALID { INVALID	Working
12	Ternary Operator	Enter a T-Operator Enter a T-Operator	?: VALID !: INVALID	Working

4 Testing With Errors

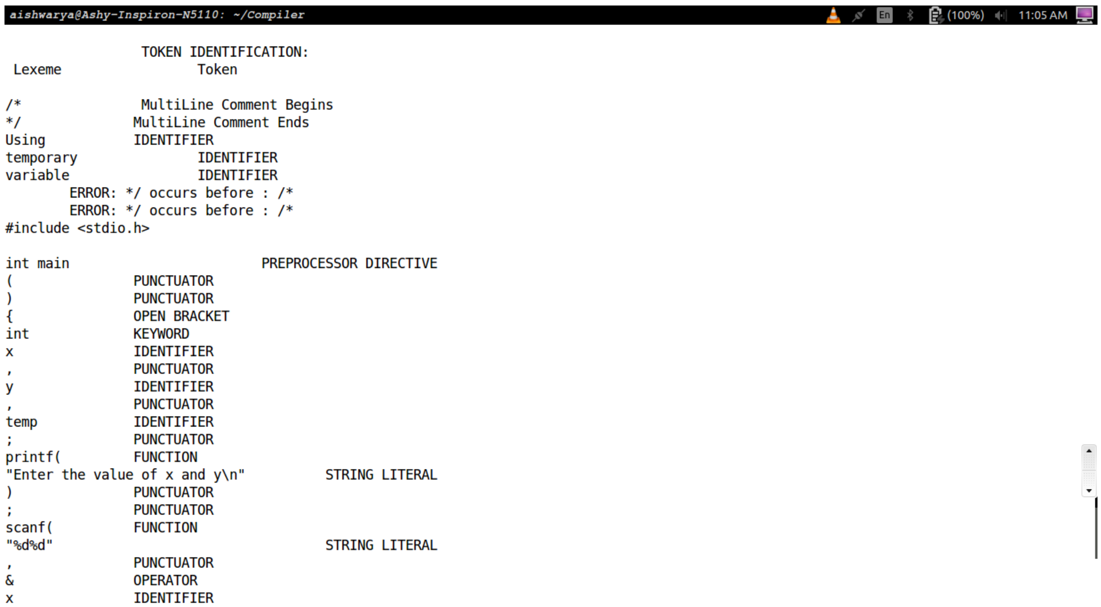


Figure 1: Errors in Nested Comments

5 Implementation

For implementation of this given code we used following technique:

5.1 To keep track of blocks

```
int comment=0,bracCount=0
comment (\/\/.*)  //line comment
comstart (\/\*)   //multi line comment starts
comend (\*\/)  //multi line comment ends
```

5.2 Keyword Definitions

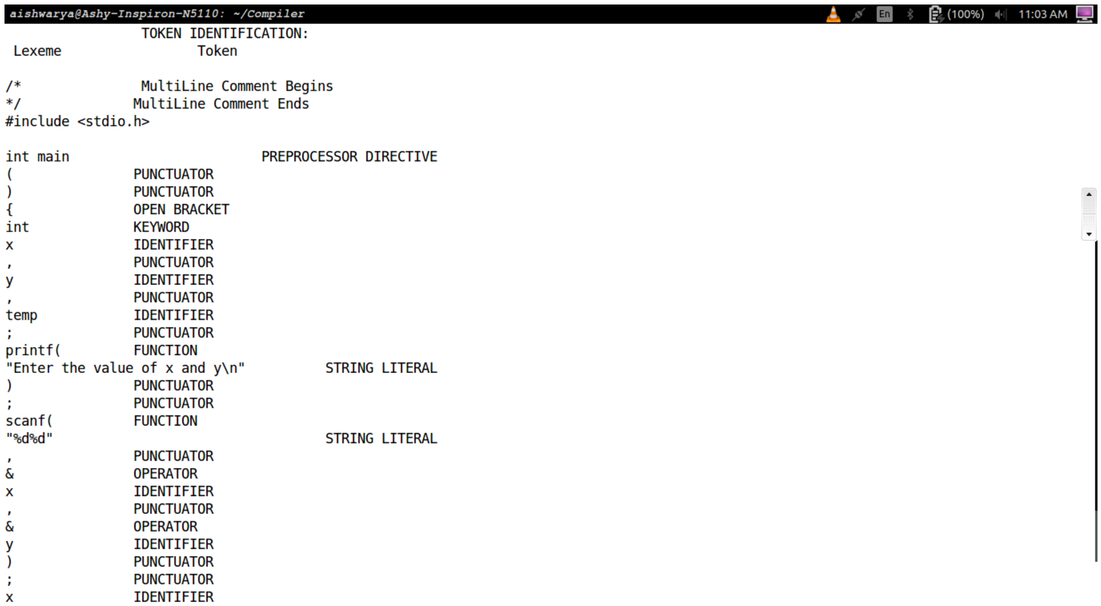
```
"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"dou
ble"|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|
"register"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"sw
```

```
itch"|"typedef"|"union"|"unsigned"|"void"|"volatile"|"while"
 relop >|<|<=|>=|!=   //Relational Operators
whitespace [ \t]+           //Whitespace
RE for a PreProcessor Statement
^#([azAZ09.]|{relop}|{whitespace})*
```

5.3 Checking for comment related errors

```
{comment}          {if(comment<=0) fprintf(yyout,"\n//Line Comment ")}
{comstart}         {comment++
                   if(comment>1)
                   fprintf(yyout,"\nERROR: Nested Comment")
                   else
                   fprintf(yyout,"\n/*ML Comment Sarts ")
          }
{comend}           {   if(comment>0)
                   {
                   comment
                   fprintf(yyout,"\n*/ML Comment Ends ")
                   }
                   else
                   fprintf(yyout,"\n\tERROR: */ found before /* ")
                   }
```

6 Results



```
aishwarya@Ashy-Inspiron-N5110: ~/Compiler                                En  (100%)  11:03 AM
                 TOKEN IDENTIFICATION:
 Lexeme                 Token

/*               MultiLine Comment Begins
*/              MultiLine Comment Ends
#include <stdio.h>

int main                        PREPROCESSOR DIRECTIVE
(               PUNCTUATOR
)               PUNCTUATOR
{               OPEN BRACKET
int             KEYWORD
x               IDENTIFIER
,               PUNCTUATOR
y               IDENTIFIER
,               PUNCTUATOR
temp            IDENTIFIER
;               PUNCTUATOR
printf(         FUNCTION
"Enter the value of x and y\n"          STRING LITERAL
)               PUNCTUATOR
;               PUNCTUATOR
scanf(          FUNCTION
"%d%d"                                  STRING LITERAL
,               PUNCTUATOR
&               OPERATOR
x               IDENTIFIER
,               PUNCTUATOR
&               OPERATOR
y               IDENTIFIER
)               PUNCTUATOR
;               PUNCTUATOR
x               IDENTIFIER
```

Figure 2: Successful Run Part 2

```
aishwarya@Ashy-Inspiron-N5110: ~/Compiler                                   (100%)   11:04 AM
y               IDENTIFIER
;               PUNCTUATOR
y               IDENTIFIER
=               OPERATOR
temp            IDENTIFIER
;               PUNCTUATOR
printf(         FUNCTION
"After Swapping\nx = %d\ny = %d\n"      STRING LITERAL
,               PUNCTUATOR
x               IDENTIFIER
,               PUNCTUATOR
y               IDENTIFIER
)               PUNCTUATOR
;               PUNCTUATOR
return          KEYWORD
0               NUMERIC CONSTANT
;               PUNCTUATOR
}               CLOSE BRACKET

 it:16, ct:2
*************Symbol Table*************
-->x
-->y
-->temp

*************************************

************Constants Table********
-->2
-->0

*************************************
aishwarya@Ashy-Inspiron-N5110:~/Compiler$
```

Figure 3: Successful Run Part 2

References

[1] En.wikibooks.org, (2016). Compiler Construction/Lexical analysis - Wikibooks, open books for an open world. [online] Available at: https://en.wikibooks.org/wiki/Compiler _Construction/Lexical _analysis [Accessed 28 Jan. 2016].

[2] Cs.fsu.edu, (2016). [online] Available at: http://www.cs.fsu.edu/ xyuan/cop5621/lect2_lexical.ppt [Accessed 28 Jan. 2016].

[3] dragonbook.stanford.edu/lecture-notes/03-Lexical-Analysis.pdf

[4] cecs.wright.edu/ tkprasad/courses/cs780/L3Lexing.pdf

[5] quex.sourceforge.net/

[6] www.cs.nyu.edu/courses/spring11/G22.2130-001/lecture4.pdf