Compiler Lab Report

Tanvir Hasan
Reg:12101005

August 12, 2015

Contents

0.1 Assignment 1: Token

0.1.1 Description

Token: In every Natural languages, there are some grammatical rules to write something. Like human language, C has some rules to write code properly. These rules are known as token.

tokens are of six types-

1. **Keywords:** A variable is a meaningful name of data storage location in computer memory. When using a variable you refer to memory address of computer. Example: do,while,for

2. **Identifiers:** The term identifier is usually used for variable names. Example: main,A,AB

3. **Constants::** Constants are expressions with a fixed value. Example: 1,23,0

4. **Strings:** Sequence of characters. Example: Ok,

5. **Special symbols:** Symbols other than the Alphabets and Digits and white-spaces. Example: (), ,

6. **Operators:** A symbol that represent a specific mathematical or non-mathematical action. Example: +, /,-,*

0.1.2 code

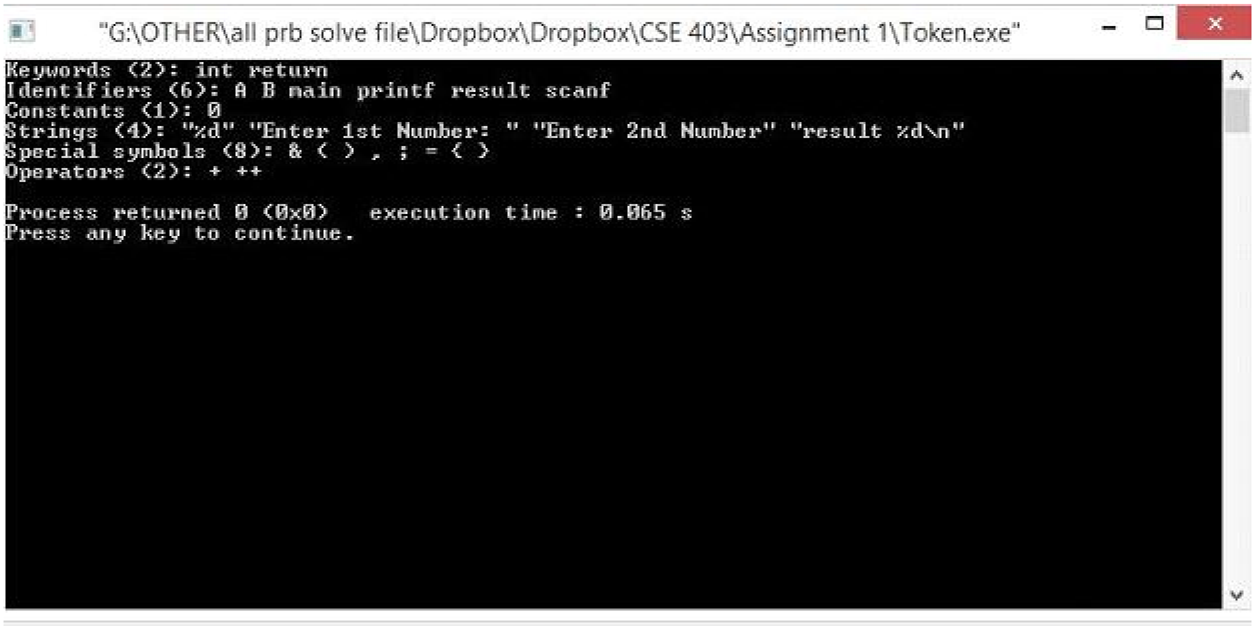
```
#include<bits/stdc++.h>
using namespace std;
template <class T> void print_vector(T v)
{
    int sz=v.size()-1;
    for(int i=0;i<=sz;i++)cout<<v[i]<<" \n"[i==sz];
}
class Token
{
    private:
        map<string,bool>token;
        map<char,bool>operators;
        string code;
        set<string>Operators,keywords,Identifiers,strings,constants;
        set<string>specialSymbol;
        void init();
        void process();
        bool isKeyWord(string str);
        bool isOperator(char str);
        bool isSpecialSymbol(char str);
    public:
        Token(string file_name);
        vector<string>getKeywords();
        vector<string>getIdentifires();
        vector<string>getStrings();
        vector<string>getConstants();
        vector<string>getOperators();
        vector<string>getSpecialSymbol();
};
void Token::init()
{
    FILE *fp=fopen("keyword.txt","r");
    while(!feof(fp))
    {
        char tk[100];
        fscanf(fp,"%s",tk);
        token[tk]=true;
    }
    fclose(fp);
    fp=fopen("operator.txt","r");
    while(!feof(fp))
    {
        char c[3];
        fscanf(fp,"%s",c);
        operators[c[0]]=true;
    }
    fclose(fp);
}
bool Token::isKeyWord(string str)
{
    return token.find(str)!=token.end();
}
bool Token::isOperator(char str)
{
    return operators.find(str)!=operators.end() and str!=' ';
}
void Token::process()
{
    for(int i=0; i<code.size(); i++)
    {
        string str="";
        if(isdigit(code[i]))
        {
            while(isdigit(code[i]))
            {
                str+=code[i];
                i++;
            }
            constants.insert(str);
            i--;
        }
        else if(code[i]=='\"')
        {
            str+='\"';
            i++;
            while(code[i]!='\"')
            {
                str+=code[i];
                i++;
            }
            str+='\"';
            strings.insert(str);
        }
        else if(isalpha(code[i]) || code[i]=='_')
        {
            while(isalpha(code[i])||isdigit(code[i])|| code[i]=='_')
            {
                str+=code[i];
                i++;
            }
            if(isKeyWord(str))keywords.insert(str);
            else Identifiers.insert(str);
            i--;
        }
        else
        {
            if(isOperator(code[i]))
            {
                while(isOperator(code[i]))
                {
                    str+=code[i];
                    i++;
                }
                Operators.insert(str);
                i--;
            }
            else if(code[i]!=' ' and code[i]!='\n')
            {
                str="";
                str+=code[i];
                specialSymbol.insert(str);
            }
        }
    }
}
Token::Token(string filename)
{
    init();
    FILE *fp=fopen(filename.c_str(),”r”);
    code=””;
    while(!feof(fp))
    {
        char c;
        fscanf(fp,”%c”,&c);
        code+=c;
    }
    fclose(fp);
    process();
}
vector<string>Token::getKeywords()
{
    vector<string>ret;
    for(auto x:keywords)ret.push_back(x);
    return ret;
}
vector<string>Token::getIdentifires()
{
    vector<string>ret;
    for(auto x:Identifiers)ret.push_back(x);
    return ret;
}
vector<string>Token::getConstants()
{
    vector<string>ret;
    for(auto x:constants)ret.push_back(x);
    return ret;
}
vector<string>Token::getStrings()
{
    vector<string>ret;
    for(auto x:strings)ret.push_back(x);
    return ret;
}
vector<string>Token::getOperators()
{
    vector<string>ret;
    for(auto x:Operators)ret.push_back(x);
    return ret;
}
vector<string>Token::getSpecialSymbol()
{
    vector<string>ret;
    for(auto x:specialSymbol)ret.push_back(x);
    return ret;
}
int main()
```

```
{
    Token t("input.cpp");
    printf("Keywords (%d): ",t.getKeywords().size());
    print_vector(t.getKeywords());
    printf("Identifiers (%d): ",t.getIdentifires().size());
    print_vector(t.getIdentifires());
    printf("Constants (%d): ",t.getConstants().size());
    print_vector(t.getConstants());
    printf("Strings (%d): ",t.getStrings().size());
    print_vector(t.getStrings());
    printf("Special symbols (%d): ",t.getSpecialSymbol().size());
    print_vector(t.getSpecialSymbol());
    printf("Operators (%d): ",t.getOperators().size());
    print_vector(t.getOperators());
    return 0;
}
```

0.1.3 Input

```
int main()
{
    int A,B;
    printf("Enter 1st Number: ");
    scanf("%d",&A);
    printf("Enter 2nd Number" );
    scanf("%d",&B);
    int result=A+B;
    A++;
    printf("result %d\n",result);
    return 0;
}
```

0.1.4 Output


"G:\OTHER\all prb solve file\Dropbox\Dropbox\CSE 403\Assignment 1\Token.exe"

```
Keywords (2): int return
Identifiers (6): A B main printf result scanf
Constants (1): 0
Strings (4): "%d" "Enter 1st Number: " "Enter 2nd Number" "result %d\n"
Special symbols (8): & ( ) , ; = { }
Operators (2): + ++

Process returned 0 (0x0)   execution time : 0.065 s
Press any key to continue.
```

0.2 Assignment 2: Postfix,Infix,Prefix

0.2.1 Description

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.

Infix notation: X + Y: Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A * (B + C) / D is usually taken to mean something like: ”First add B and C together, then multiply the result by A, then divide by D to give the final answer.”

Postfix notation (also known as ”Reverse Polish notation”): X Y + : Operators are written after their operands. The infix expression given A * (B + C) / D is equivalent to A B C + * D / .

Prefix notation (also known as ”Polish notation”): + X Y : Operators are written before their operands. The expressions A * (B + C) / D are equivalent to / * A + B C D .

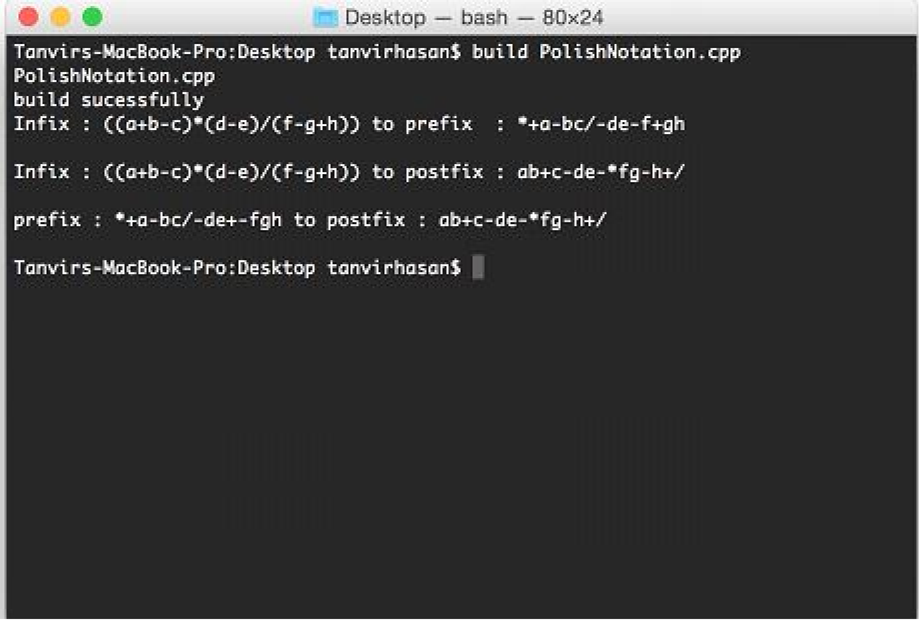
0.2.2 Code

```
#include<bits/stdc++.h>
using namespace std;
int get_operator_priority(char c)
{
    int p = 0;
    switch (c)
    {
    case '*':
    case '/':
        return 2;
    case '+':
    case '-':
        return 1;
    default:
        return 0;
    }
}
class PolishNotation
{
    private:
        string Expre;
        string prefixToInfix();
    public:
        PolishNotation(string str);
        string infixToPostfix();
        string infixToprefix();
        string prefixToPostfix();
};

```

```
PolishNotation::PolishNotation(string str)
{
    Expre=str;
}

string PolishNotation::infixToPostfix()
{
    stack<char>Stk;
    string ret="";
    for(int i=0;i<Expre.size();i++)
    {
        char curr=Expre[i];
        if(isalpha(curr))ret+=curr;
        else if(curr=='(')Stk.push(curr);
        else if(curr==')')
        {
            char d=Stk.top();
            Stk.pop();
            while(d!='(')
            {
                ret+=d;
                d=Stk.top();
                Stk.pop();
            }
        }
        else
        {
            while(!Stk.empty() and get_operator_priority(Stk.top())>= get_operator_priority(curr))
            {
                ret+=Stk.top();
                Stk.pop();
            }
            Stk.push(curr);
        }
    }
    while(!Stk.empty())
    {
        ret+=Stk.top();
        Stk.pop();
    }
    return ret;
}

string PolishNotation::infixToprefix()
{
    stack<char>Stk;
    string ret="";
    for(int i=Expre.size()-1;i>=0;i--)
    {
        char curr=Expre[i];
        if(isalpha(curr))ret+=curr;
        else if(curr==')')Stk.push(curr);
        else if(curr=='(')
        {
            char d=Stk.top();
            Stk.pop();
            while(d!=')')
            {
                ret+=d;
                d=Stk.top();
                Stk.pop();
            }
        }
        else
        {
            while(!Stk.empty() and get_operator_priority(Stk.top())>= get_operator_priority(curr))
            {
                ret+=Stk.top();
                Stk.pop();
            }
            Stk.push(curr);
        }
    }
    while(!Stk.empty())
    {
        ret+=Stk.top();
        Stk.pop();
    }
    reverse(ret.begin(),ret.end());
    return ret;
}

string PolishNotation::prefixToInfix()
{
    stack<string>Stk;
    for(int i=Expre.size()-1;i>=0;i--)
    {
        char curr=Expre[i];
        if(isalpha(curr))Stk.push(string("")+curr);
        else
        {
            string A=Stk.top();
            Stk.pop();
            string B=Stk.top();
            Stk.pop();
            string C="";
            if(curr=='+' or curr=='-')C=A+curr+B;
            else if(curr=='/')C="("+A+")"+curr+"("+B+")";
            else if(curr=='*')C="("+A+")"+curr+B;
            Stk.push(C);
        }
    }
    return Stk.top();
}

string PolishNotation::prefixToPostfix()
{
    Expre=prefixToInfix();
    cout<<Expre<<endl;
    return infixToPostfix();
```

```
}
int main()
{
    string infix="((a+b-c)*(d-e)/(f-g+h))";
    string prefix="*+a-bc/-de+-fgh";
    PolishNotation Infix(infix);
    cout<<"Infix : "<<infix<<" to prefix  : "<<Infix.infixToprefix()<<endl<<endl;
    cout<<"Infix : "<<infix<<" to postfix : "<<Infix.infixToPostfix()<<endl<<endl;
    PolishNotation Prefix("AB+CD+*");
    cout<<"prefix : "<<prefix<<" to postfix : "<<Prefix.prefixToPostfix()<<endl<<endl;
    return 0;
}
```

0.2.3 Output:



```
Tanvirs-MacBook-Pro:Desktop tanvirhasan$ build PolishNotation.cpp
PolishNotation.cpp
build sucessfully
Infix : ((a+b-c)*(d-e)/(f-g+h)) to prefix  : *+a-bc/-de-f+gh

Infix : ((a+b-c)*(d-e)/(f-g+h)) to postfix : ab+c-de-*fg-h+/

prefix : *+a-bc/-de+-fgh to postfix : ab+c-de-*fg-h+/

Tanvirs-MacBook-Pro:Desktop tanvirhasan$ 
```

0.3 Assignment 3: Flex

0.3.1 Description

Flex is a tool for generating scanners. A scanner, sometimes called a tokenizer, is a program which recognizes lexical patterns in text. The flex program reads user-specified 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 a C source file named, ”lex.yy.c”, which defines the function yylex(). The file ”lex.yy.c” can be compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding C code.

0.3.2 Identification of C Token:

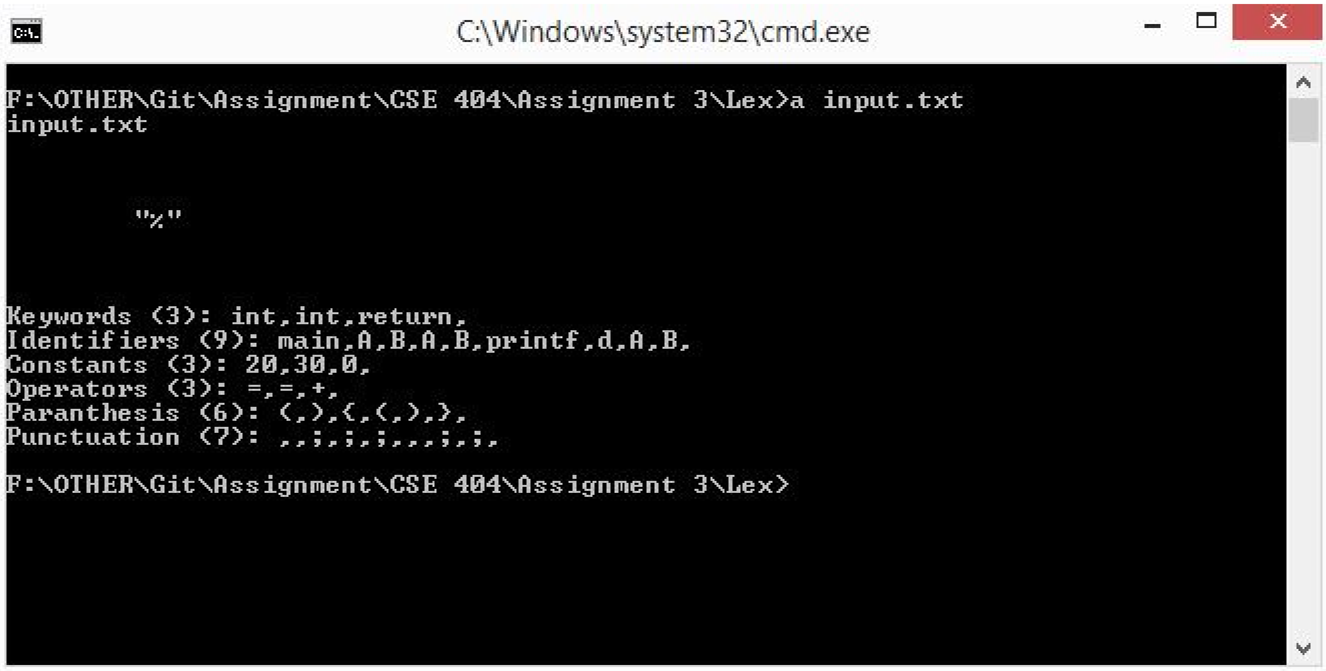
Code

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

   int TotalKeyword=0;
   int TotalIdentifier=0;
   int TotalOperator=0;
   int TotalConstant=0;
   int TotalPunctuation=0;
   int TotalParenthesis=0;

   char KeywordList[1000];
   char IdentifierList[1000];
   char OperatorList[1000];
   char ConstantList[1000];
   char PunctuationList[1000];
   char ParenthesisList[1000];
%}

token auto|break|c(har|co(nst|ntinue))|do(uble)?|e(lse|num|xtern)|f(loat|or)|i(f|nt)|long|re(gister|turn)|s(i(gned|zeof)|t(atic|ruct)|witch)|typedef|unsigned|vo(id|latile)|while|_Packed
identifier [a-zA-Z][a-zA-Z0-9]*
op [-|+|\*|/|^|=]
constant [0-9]+
paranthesis [{|}|\[|\]|\(|\)]
punctuation [;|:|,]

%%
{token} {
   TotalKeyword++;
   strcat(KeywordList, yytext);
   strcat(KeywordList, ",");
}
{identifier} {
   TotalIdentifier++;
   strcat(IdentifierList, yytext);
  strcat(IdentifierList, ",");
}
{constant} {
  TotalConstant++;
  strcat(ConstantList, yytext);
  strcat(ConstantList, ",");
}
{paranthesis} {
  TotalParenthesis++;
  strcat(ParenthesisList, yytext);
  strcat(ParenthesisList, ",");
}
{op} {
  TotalOperator++;
  strcat(OperatorList, yytext);
  strcat(OperatorList, ",");
}
{punctuation} {
  TotalPunctuation++;
  strcat(PunctuationList, yytext);
  strcat(PunctuationList, ",");
}


%%

int main(int argc, char **argv){
  printf("%s",argv[1]);
  FILE *fp=fopen(argv[1],"r");
  yyin = fp;
  yylex();
  printf("\n\nKeywords (%d): %s \n", TotalKeyword, KeywordList);
  printf("Identifiers (%d): %s \n", TotalIdentifier, IdentifierList);
  printf("Constants (%d): %s \n", TotalConstant, ConstantList);
  printf("Operators (%d): %s \n", TotalOperator, OperatorList);
  printf("Paranthesis (%d): %s \n", TotalParenthesis, ParenthesisList);
  printf("Punctuation (%d): %s \n", TotalPunctuation, PunctuationList);
  return 0;
}

int yywrap(void){
  return 1;
}

int yyerror(void){
  printf("Error\n");
  exit(1);
}
```

Input

```
int main(){
  int A,B;
  A=20;
  B=30;
  printf("%d",A+B);
  return 0;
}
```

Output



```
F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>a input.txt
input.txt



        "%"



Keywords (3): int,int,return,
Identifiers (9): main,A,B,A,B,printf,d,A,B,
Constants (3): 20,30,0,
Operators (3): =,=,+,
Paranthesis (6): (,),{,(,),},
Punctuation (7): ,,;,;,;,,,;,;,

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>
```

0.3.3 Convert Roman numbers to Decimal numbers

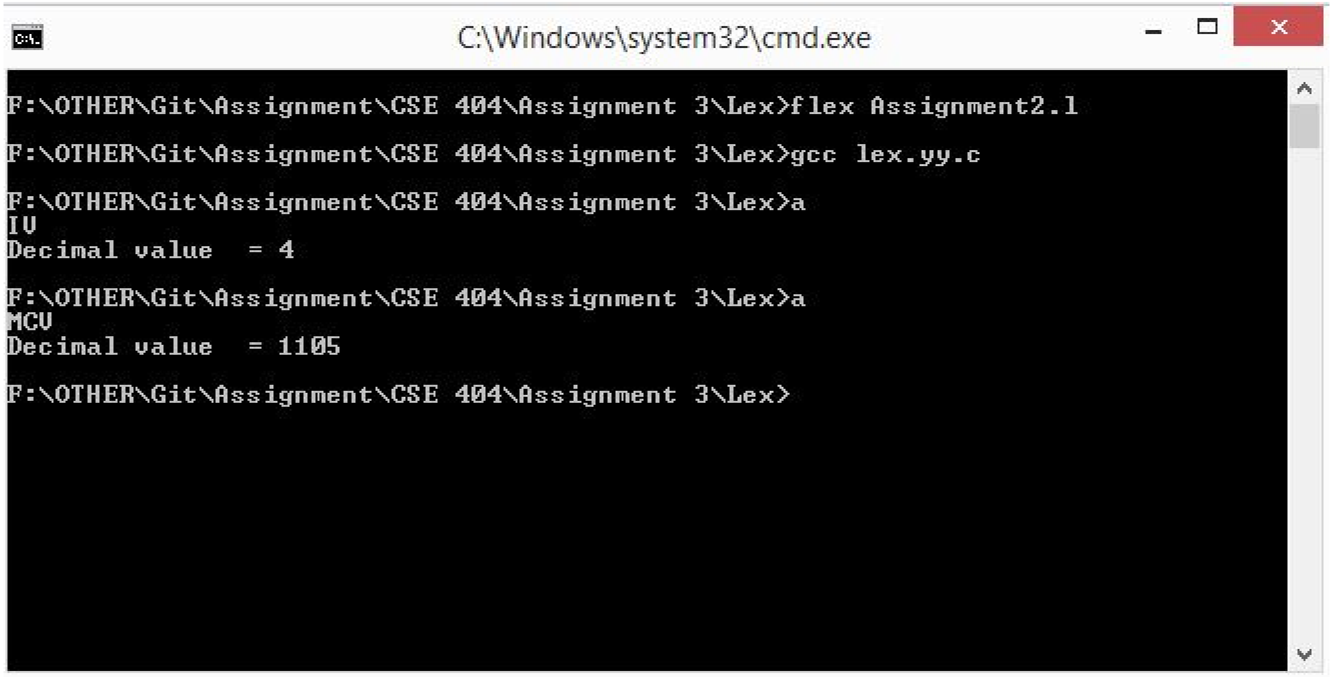
Code

```
%{
  int sum=0;
%}

%%

I {
  sum += 1;
}
IV {
  sum += 4;
}
V {
  sum += 5;
}
IX {
  sum += 9;
}
X {
  sum += 10;
}
XL {
  sum += 40;
}
L {
  sum += 50;
}
XC {
  sum += 90;
}
C {
  sum += 100;
}
CD {
  sum += 400;
}
D {
  sum += 500;
}
CM {
  sum += 900;
}
M {
  sum += 1000;
}

[\n] {
  return sum;
}

%%
int main (void) {

  yylex ();

  printf (”Decimal value  = %d\n”, sum);
  return 0;
}

int yywrap(void){
  return 0;
}

int yyerror(void){
  printf(”Error\n”);
  exit(1);
}
```

Input

IV
MCV

Output



```
F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>flex Assignment2.l

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>gcc lex.yy.c

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>a
IV
Decimal value  = 4

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>a
MCV
Decimal value  = 1105

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>
```

0.3.4 Identification of various number formats

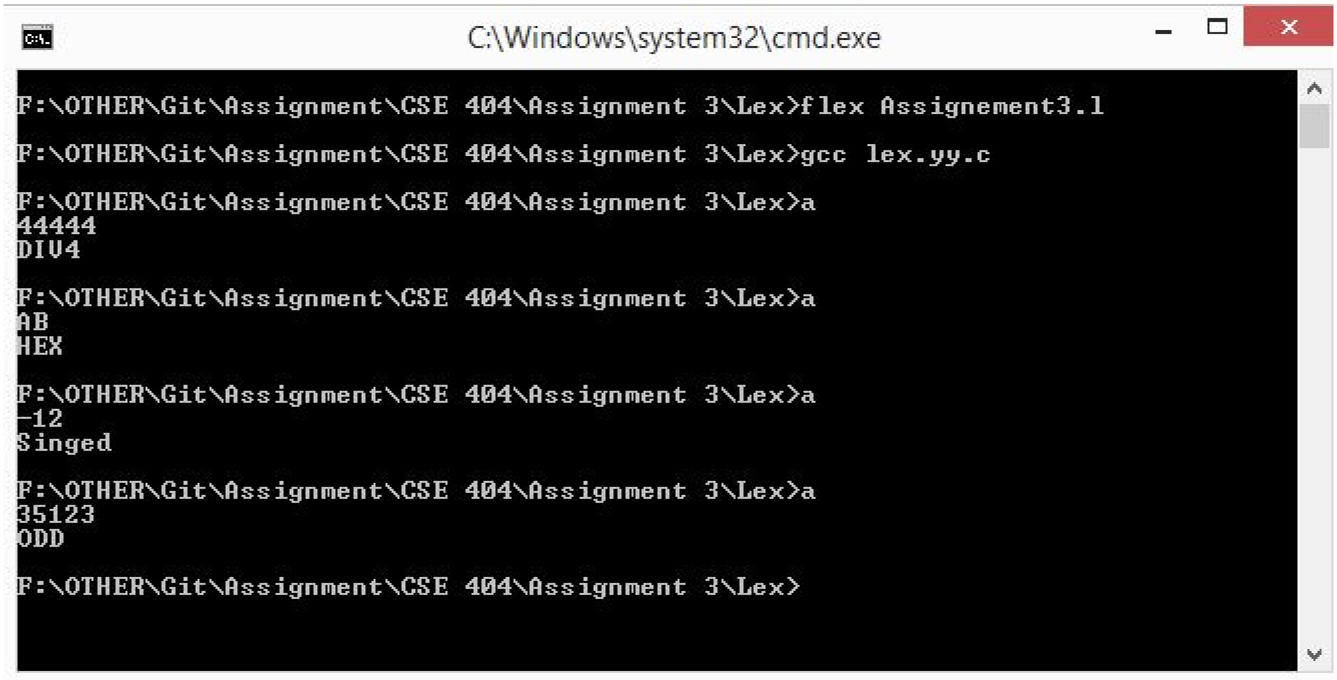
Code

```
%{
  #include<stdio.h>
  #include<stdlib.h>
  int pos  = 0;
  int result[100];
  char Ans[][30]={" ","ODD","DIV4","Singed","Decimal","Scientefic","HEX","Overflow",
    "String","Identifier","Unknown"};
%}

digit [0-9]{4,}
decimal [0-9]+\.[0-9]+
singed (-|\+)[0-9]+
hex [0-9A-F]+
string [0-9][0-9a-zA-Z]*
scientefic ({singed}(e|E){singed})
identifier [a-zA-Z][0-9a-zA-Z]*
unknown [^(-\+0-9a-zA-Z \n)]

%%
[ ] {}
{digit} {
  int number=atoi(yytext);
  int len=yyleng;
  if(number%2){
    result[pos]=1;
  }else if(number%4==0){
    result[pos]=2;
  }else if(len>4){
    result[pos]=7;
  }else{
    result[pos]=10;
  }
  pos++;
}
{singed} {
  result[pos]=3;
  pos++;
}
{decimal} {
  result[pos]=4;
  pos++;
}
{scientefic} {
  result[pos]=5; pos++;
}
{hex} {
  int len=yyleng;
  if(len>4){
    result[pos]=7;
  }else{
    result[pos]=6;
  }
  pos++;
}
{string} {
  result[pos]=8; pos++;
}
{identifier} {
  result[pos]=9; pos++;
}
{unknown} {
  result[pos]=10; pos++;
}
[\n] {
  return 0;
}
%%


int main(int argc, char **argv){
  yyin = stdin;
  yylex();
  int i=0;
  for(i=0;i<pos;i++){
    printf("%s\n",Ans[result[i]]);
  }
  return 0;
}

int yywrap(void){
  return 0;
}
```

Input

4444
AB
-12
35123

Output



```
F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>flex Assignement3.l

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>gcc lex.yy.c

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>a
44444
DIV4

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>a
AB
HEX

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>a
-12
Singed

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>a
35123
ODD

F:\OTHER\Git\Assignment\CSE 404\Assignment 3\Lex>
```