Lexical Scanner

Team Members : Arvind Sai K (15CO207) and Derik Clive (15CO213)

Pipeline for execution of a program

The following image shows the typical flow for an execution of a program starting from the raw source code stage.

What is a compiler?

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). Compilers are a type of translator that support digital devices, primarily computers. The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

Introduction to compilers

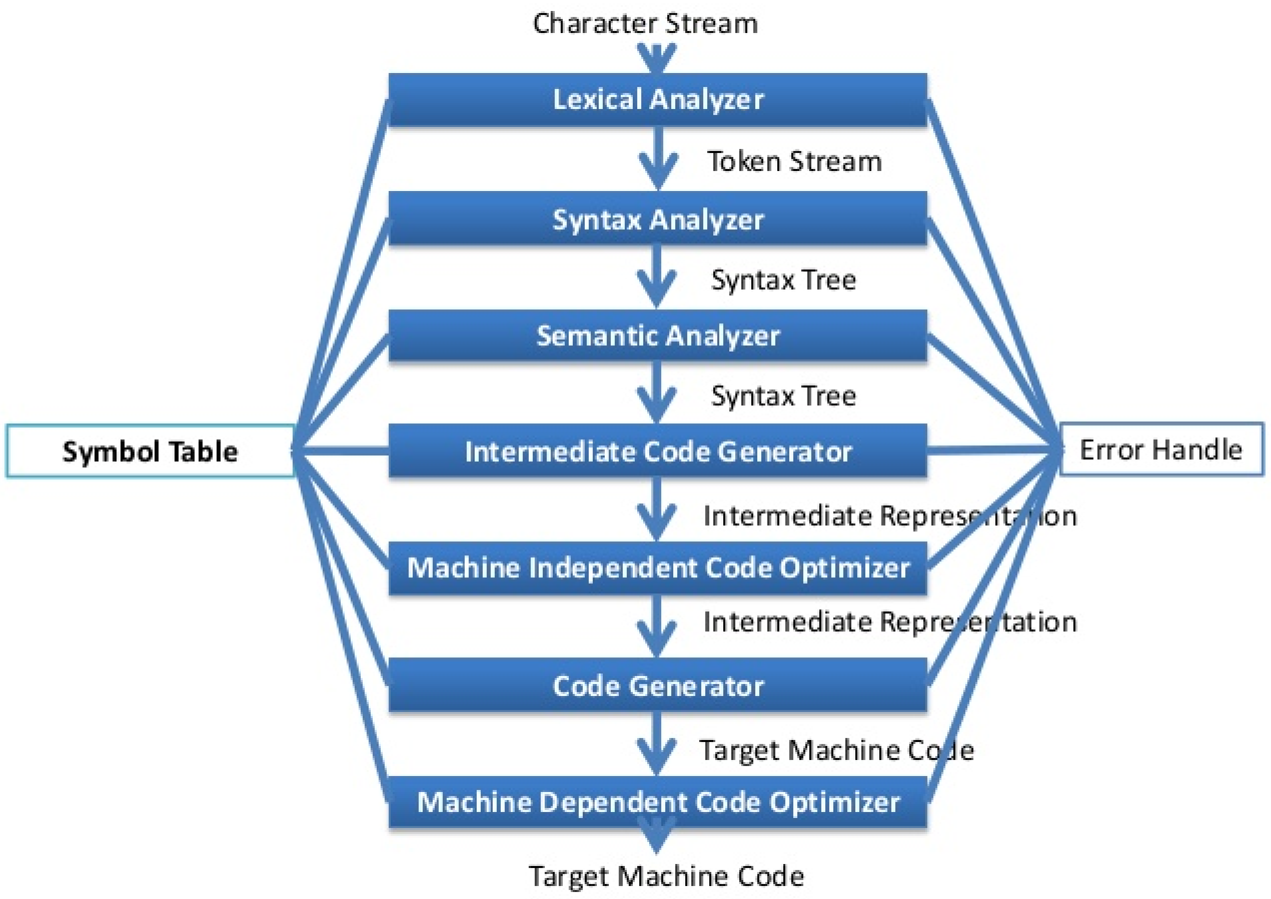
Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to cre- ate an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semanti- cally unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

The synthesis part constructs the desired target program from the interme- diate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.

If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in figure below. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise pro- duced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in figure below may be missing.



Analysis phase in compilers

Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as below.

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Synthesis phase in compilers

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table (Common to all the above phases)

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Details of the lexical analysis phase

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Lexical analyzers are divided into a cascade of two processes:

1. Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
2. Lexical analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

When discussing lexical analysis, we use three related but distinct terms:

1. A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.
2. A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.
3. A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Lex code for lexical analyser

```
%x comment
%x string_literal
%{
    #include<stdio.h>

    #define KNRM  "\x1B[0m"
    #define KRED  "\x1B[31m"
    #define KGRN  "\x1B[32m"
    #define KYEL  "\x1B[33m"
    #define KBLU  "\x1B[34m"
    #define KMAG  "\x1B[35m"
    #define KCYN  "\x1B[36m"
    #define KWHT  "\x1B[37m"

   #define n_buckets 1000
   int pstack[100];
   int ptop=-1;
   int cstack[100];
   int ctop=-1;
   int line_num = 1;
   int nested_comment_stack=0;
   char token[100];

   struct table_entry
   {
      void *key, *value;
      struct table_entry *next;
      unsigned int line;
   };

   struct table_entry *s_head[n_buckets];
   struct table_entry *c_head[n_buckets];

   void install_symbol();
   void install_constant();

%}

identifier [a-zA-Z_]([a-zA-Z0-9])*
digit [0-9]
BID     ([0-9]|!|@|#|$|%)+([a-zA-Z0-9])+

escape_sequence [a|n|b|t|f|r|v|\|"|'|?]
white_space [ \t]
backslash [\]
double_quotes ["]

%%

\n {yylineno++;}
{white_space}*

#include[ ]*<[^>]+> {printf("%s\n%40s%40s%40d", KBLU,
yytext,"Preprocessor-directive", yylineno);}
printf {printf("%s\n%40s%40s%40d", KBLU,"printf", "Pre-defined function",
yylineno);strcpy(token, "function");install_symbol();}
scanf {printf("%s\n%40s%40s%40d", KBLU,"scanf", "Pre-defined function",
yylineno);strcpy(token, "function");install_symbol();}
"/*"                    {BEGIN(comment); nested_comment_stack=1; yymore();}
<comment><<EOF>>        {printf("\nMulti-line Comment: \""); yyless(yyleng-2); ECHO;
printf("\", not terminted."); yyterminate();}
<comment>"/*"           {nested_comment_stack++; yymore();}
<comment>.              {yymore();}
<comment>\n             {yymore();yylineno++;}
<comment>"*/"           {nested_comment_stack--;
                        if(nested_comment_stack<0)
                        {
                          printf("\nComment: \"%s\", not balanced at line no: %d.",
yytext, yylineno);
                          yyterminate();
                        }
                        else if(nested_comment_stack==0)
                        {
                          /*printf("\nMulti-line comment : \"%s\" at line number:
%d.", yytext, yylineno);*/
                          BEGIN(INITIAL);
                        }
                        else
                          yymore();
                        }

"*/"                    {printf("\n Uninitialised comment at line number: %d.", 
yylineno); yyterminate();}

"//".*                  {printf("\nSingle-line comment : \"%s\" at line number: 
%d.", yytext, yylineno);}



<INITIAL>{double_quotes}                        { BEGIN(string_literal); yymore();}
<string_literal>"\\"+{escape_sequence}              {printf("%s\n%40s%c%39s%40d", 
KBLU,  "\\", yytext[yyleng-1],"Escape Sequence", yylineno); 
                                                yymore();}
<string_literal>"\\"+[^a|n|b|t|f|r|v|\|"|'|?]           {printf("\nUnrecognized 
escape seqence at line number: %d.", yylineno);}
<string_literal>{double_quotes}                 {printf("%s\n%40s%40s%40d",KBLU, 
yytext, "String Constant",yylineno); 
                                                         strcpy(token, "String 
Constant");  install_constant(); BEGIN(INITIAL);}

<string_literal>\n                              {printf("\nError : Unterminated 
string: %s at line number: %d.", yytext, yylineno);yylineno++; BEGIN(INITIAL);}
<string_literal>[^\\]                               {yymore();}

{digit}+    {printf("%s\n%40s%40s%40d", KBLU, yytext, "Integer Constant", 
yylineno); strcpy(token, "INT Constant");  install_constant();}
{digit}*\.?{digit}*(E[+|-]?{digit}+*\.?{digit}*)?   {printf("%s\n%40s%40s%40d", 
KBLU, yytext, "Floating Point Constant",yylineno); strcpy(token, "FP Constant"); 
install_constant();}
{digit}*\.?{digit}*E.?  {printf("%s\nError No exponent provided: %s , line number: 
%d.", KBLU, yytext,yylineno);}
\'.\'   {printf("%s\n%40s%40s%40d", KBLU, yytext, "Character Constant",yylineno); 
strcpy(token, "Char Constant");  install_constant();}

^{white_space}*(unsigned|signed)?(void|int|char|short|long|float|double){white_spac
e}+{identifier}{white_space}*\([^)]*\){white_space}*  {

if(strstr(yytext, "main")!=NULL)

{

printf("%s\n%40s%40s%40d", KBLU, "main", "Main Function", yylineno);

strcpy(token, "Main function");


}

else

{

printf("%s\n%40s%40s%40d", KBLU,  yytext, "User-defined function", yylineno);
```

```
strcpy(token, "User Defined function");


}

install_symbol();

}


"auto"                      {printf("%s\n%40s%40s%40d", KBLU, yytext, "Keyword",
yylineno); strcpy(token,"Keyword");  install_symbol();}
"break"                                 {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"case"                              {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"char"                                  {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"const"                                     {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"continue"                                  {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"default"                                   {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"do"                                        {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"double"                                    {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"else"                                      {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"enum"                                  {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"extern"                                {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"float"                                 {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"for"                                       {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"goto"                                      {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"if"                                        {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"int"                                       {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"long"                                      {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"register"                          {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"return"                                {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"short"                                 {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"signed"                                    {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"sizeof"                                    {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
```

```
"static"                                    {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"struct"                                {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"switch"                                {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"typedef"                               {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"union"                                     {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"unsigned"                              {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"void"                              {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"volatile"                              {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}
"while"                                     {printf("%s\n%40s%40s%40d", KBLU,
yytext, "Keyword", yylineno);  strcpy(token,"Keyword");  install_symbol();}



{identifier}                            {printf("%s\n%40s%40s%40d", KBLU,  yytext,
"Identifier", yylineno); strcpy(token,"Identifier"); install_symbol();}


{BID}           {printf("%s\n%40s%40s%40d", KRED, yytext, "Invalid Identifier",
yylineno); }


^{white_space}*(unsigned|signed)?(void|int|char|short|long|float|double){white_spac
e}+\*+[ ]*{identifier}                 {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Pointer Declration", yylineno);}
{identifier}+\[{digit}*\]        {printf("%s\n%40s%40s%40d", KBLU, yytext, "Array
Declaration", yylineno);}

[\(]        {if(ptop==-1)
              {
                ptop=0;
                pstack[ptop] = yylineno;
              }
              else
              {
                ptop++;
                pstack[ptop] = yylineno;
              }
              printf("%s\n%40s%40s%40d", KBLU, yytext, "Operator",yylineno);
            }

[\)]      {
            if(ptop==0)
             {
               ptop=-1;
             }
             else
             {
                ptop--;
             }
```

```
                printf("%s\n%40s%40s%40d", KBLU, yytext, "Operator",yylineno);
            }

[\{]         {if(ctop==-1)
                {
                  ctop=0;
                  cstack[ctop] = yylineno;
                }
                else
                {
                  ctop++;
                  cstack[ctop] = yylineno;
                }
                printf("%s\n%40s%40s%40d", KBLU, yytext, "Operator",yylineno);
              }

[\}]        {
              if(ctop==0)
               {
                 ctop=-1;
               }
               else
               {
                  ctop--;
               }
                printf("%s\n%40s%40s%40d", KBLU, yytext, "Operator",yylineno);
            }

(\+|-|\*|\/|\&|\[|\]|\>|\<|!=|\+\+|--|\%|>=|<=|==|&&|\|\||!|\+=|-=|\/=|\*=|\%=|\&=|
\|=|\^=|\.|\-\>|=|\{|\}|\))  {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Operator",yylineno); }
[,]                                 {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Separator", yylineno);}
[;]                                 {printf("%s\n%40s%40s%40d", KBLU, yytext,
"Delimiter",yylineno);}
\'.                                 {printf("%s\nUnterminated CHARACTER LITERAL: %s,
\tline no:%d\n",KRED, yytext, yylineno);}
.                                   {printf("%s\n%40s%40s%40d", KRED, yytext, "Invalid
Character", yylineno);}
%%

unsigned int get_hash(char *str)
{
    unsigned int hash = 5381;
    int c;

    while ((c = *str++))
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash%1000;
}

struct table_entry *create_node()
{
  struct table_entry *temp = (struct table_entry *)malloc(sizeof(struct
table_entry));
  if(temp==NULL)
  {
    printf("\nCould not allocate memory for the symbol table.");
    exit(1);
  }
  temp->next = NULL;
  return temp;
}

void insert(struct table_entry *head[], unsigned int index, void *key, void *value,
unsigned int line)
{
  struct table_entry *temp = create_node();
  temp->key = key;
  temp->value = value;
  temp->line = line;
  temp->next = head[index];

  head[index] = temp;
}

struct table_entry *search(struct table_entry *head, void *key)
{
  struct table_entry *temp = head;
  while(temp!=NULL)
  {
    if(strcmp((char *)temp->key, (char *)key)==0)
      return temp;
    temp = temp->next;
  }
  return temp;
}



void install_symbol()
{
  char *key = (char *)malloc(sizeof(char)*yyleng);
  char *value = (char *)malloc(sizeof(char)*yyleng);

  strcpy(key, yytext);
  strcpy(value, token);
  unsigned int index = get_hash(key);


  struct table_entry *temp = search(s_head[index], key);
  if(temp==NULL)
    insert(s_head, index, key, value, yylineno);
}

void install_constant()
{
  char *key = (char *)malloc(sizeof(char)*yyleng);
  char *value = (char *)malloc(sizeof(char)*yyleng);

  strcpy(key, yytext);
  strcpy(value, token);
  unsigned int index = get_hash(key);
  struct table_entry *temp = search(c_head[index], key);
  if(temp==NULL);
    insert(c_head, index, key, value, yylineno);
}


void print_symbol_table()
{
  int i;
  char a[100]="<";

printf("%s\n======================================================================
==================================================================================
====", KRED);
  printf("%s\n\t\t\t\t\t\t\t\t\tSYMBOL TABLE", KBLU);

printf("%s\n======================================================================
==================================================================================
====", KRED);
  printf("%s\n%40s%40s%40s", KCYN, "TOKEN", "TOKEN TYPE", "LINE NUMBER");
  for(int i=0;i<n_buckets;i++)
  {
      if(s_head[i]!=NULL)
      {
        struct table_entry *temp = s_head[i];
        while(temp!=NULL)
        {
          printf("%s\n%40s%40s>%40d", KWHT, (char *)temp->key, strcat(a, (char
*)temp->value), temp->line);
          strcpy(a, "<");
          temp = temp->next;
        }
      }
  }
  printf("\n");
}

void print_constant_table()
{int i;
  char a[100]="<";

printf("%s\n======================================================================
==================================================================================
====", KRED);
  printf("%s\n\t\t\t\t\t\t\t\t\tCONSTANT TABLE", KBLU);

printf("%s\n======================================================================
==================================================================================
====", KRED);
  printf("%s\n%40s%40s%40s", KCYN, "TOKEN", "TOKEN TYPE", "LINE NUMBER");
  for(int i=0;i<n_buckets;i++)
  {
      if(c_head[i]!=NULL)
      {
        struct table_entry *temp = c_head[i];
        while(temp!=NULL)
        {
          printf("%s\n%40s%40s>%40d", KWHT, (char *)temp->key, strcat(a, (char
*)temp->value), temp->line);
          strcpy(a, "<");
          temp = temp->next;
        }
      }
  }
  printf("\n");
}


int main()
{
  FILE *fp;
  fp = fopen("sample.c", "r");
  yyin = fp;

printf("\n===========================================================================
=====================================================================================
");
  printf("\n%40s%40s%40s", "TOKEN", "TOKEN TYPE", "LINE NUMBER");

printf("\n===========================================================================
=====================================================================================
");
  int newtoken = 1;
  while(newtoken){
    newtoken = yylex();
  }
  if(ptop!=-1)
  {
    printf("\n\n\t\t\'(\' has not been matched at line number %d.", pstack[ptop]);
  }
  if(ctop!=-1)
  {
    printf("\n\n\t\t\'{\' has not been matched at line number %d.", cstack[ctop]);
  }
  print_symbol_table();
  print_constant_table();
  return 0;
}
int yywrap()
{
  return 1;
}
```

Test cases

Test case 1

1) Test for identifying int and char data types and their corresponding sub-types like short , long , signed, unsigned.

2) Test for identifying while and nested while constructs

```
#include <stdio.h>

/* 1 ) Test for identifying int and char data types and their corresponding
sub-types
like short , long , signed, unsigned.
2 ) Test for identifying while and nested while constructs  */
int compu(int a)
{

}
int main(){
    /* test for various integer types supported */
    short int var1;
    long int var2;
    long long int var3;
    int var4;
    int $cd;
    signed short int var5;
    signed long int var6;
    signed long long int var7;
    signed int var8;
    unsigned short int var5;
    unsigned long int var6;
    unsigned long long int var7;
    unsigned int var8;

    /* test for various character types supported */
    char var9 != 'b';
    signed char var10;
    signed char var11;
    float var12 = 9.56;
    /* test for while and nested while */
    var1 = 0;
    while(var1 < 20){
        var2 = 0;
        while(var2 < 40){
            var3 = 0;
            var2 = var2 + 1;
        }
        var1 = var1 + 1;
    }

    var1 = 0;
    while(var1 < 20){
        var2 = 0;
        var1 = var1 + 1;
    }
    printf("\nDone\n");
    return 0;
}
```

Output 1

Test case 2

1) Test case for identifying function with single argument

2) Test for identifiers and constants supported

3) Test for strings and special symbols supported

```
#include <stdio.h>

/* 1 ) Test case for identifying function with single argument
2 ) Test for identifiers and constants supported
3 ) Test for strings and special symbols supported
*/


/* Test case for identifying function with single argument */
int power2(int c){
    int d = c*c;
    return d;
}
char add1(char c){
    return (c+1);
}
void starter(int a){
    printf("you wanted to print %d",a);
}
int main(){
    /* test for identifiers and constants supported */
    short int sum = 10;
    long int total = 20;
    sum = 10*10 + 20;

    /* test for strings and special symbols supported */
    char a[100] = "hello";
    printf("Hello world");

    int ab[2] = {10,20};
    int b = 3;
    b = (10 + b)*2 - 3;

    int res1 = power2(10);
    char res2 = add1('d');
    starter(20);
    return 0;
}
```

Output 2

Test case 3

1) Test case for identifying escape sequences

2) Test for some valid multiline comments

3) Test for pointers

```
#include <stdio.h>

/* 1 ) Test case for identifying escape sequences
2 ) Test for some valid multiline comments
3 ) Test for pointers
*/

int main(){
    /* Test case for identifying escape sequences */
    printf("testing \t escape \n sequences \n");

    /* Test for some valid multiline comments  */

    /* Nested /*
    Multiline comm
    ents work */

    /* Test for pointers */
    char c = 'a';
    char * temp = &c;
    return 0;
}
```

Output 3

Test case 4

1) Test case for string not terminated

2) Test for unbalanced paranthesis;

3) Test for stray characters

4) Multiline comment not terminated

```
#include <stdio.h>

/* 1 ) Test case for string not terminated
2 ) Test for unbalanced paranthesis;
3 ) Test for stray characters
4 ) Multiline comment not terminated
*/

int main(){
    printf("hi there);

    int a = 0;
    int b = 3;
    int c = 5;

    a = ((b+c*a);

    ```
    a = 3;

    return 0;
}

/*
    this comment does
    not end
```

Output 4

Test case 5

1) Test for '{' not terminated

2) Test for unterminated character constant

3) Test for invalid functions

```
#include <stdio.h>

/* 1 ) Test for '{' not terminated
2 ) Test for unterminated character constant
3 ) Test for invalid functions
*/

int func1(int a) //Valid function
{
    return 0;
}

void func2(int a, float int b) //Valid Function
{
    int var1;
}

void func3(int a int b) //Invalid Function
{
    int var2;
}

void func4(short int a, b) //Invalid Function
{
    int var3
}
```
```

```
int main(){

   char a = 'a;
   {
       int var5;
       {
           int var4;
       }

    return 0;

}

/*
    this comment does
    not end
```

Output 5