**Project 1: Scanner for C-language**

**Lexical Analysis**

By: Saumyadip Mandal (15CO144)

Shishir Gangwar     (15CO147)

A **compiler** is [computer software](https://en.wikipedia.org/wiki/Computer_software) that transforms computer code written in one [programming language](https://en.wikipedia.org/wiki/Programming_language) (the source language) into another programming language

(the target language). Compilers are a type of [translator](https://en.wikipedia.org/wiki/Translator_(computing)) that support digital devices, primarily computers. The name compiler is primarily used for programs that translate [source code](https://en.wikipedia.org/wiki/Source_code) from a [high-level programming language](https://en.wikipedia.org/wiki/High-level_programming_language) to a [lower level language](https://en.wikipedia.org/wiki/Lower_level_language) (e.g., [assembly language](https://en.wikipedia.org/wiki/Assembly_language), [object code](https://en.wikipedia.org/wiki/Object_code), or [machine code](https://en.wikipedia.org/wiki/Machine_code)) to create an [executable](https://en.wikipedia.org/wiki/Executable) program.

   A compiler operates in two main phases, each of which transforms the source program from one representation to another. For compilation of any source program written in some source language, the compiler passes from Analysis and Synthesis phases.

**Analysis Phase**: Analyse source program and build an intermediate representation.

**Synthesis Phase**: Generate target program from intermediate representation.

Analysis Of Source Program Include:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis

**Lexical Analysis**

Lexical analysis is the first phase of the compiler. It receives the input from the source program and produces tokens as output. Starting from left to right it reads the characters one by one and forms the tokens. Token usually represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

Example: a + b = 20

Here, a, b, +, =, 20 are all represents separate tokens. A group of characters forming a token is called the Lexeme. The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.

Another important task of the lexical analyser is to build a symbol table. Symbol table is a table of all the identifiers (such as variable names, procedures, and constants) used in the program. When an identifier is first recognized by the analyser, it is inserted into the symbol table in addition with information about its type, where it is to be stored, and so forth. This information is used in subsequent passes of the compiler.

For example,

**Dist = 0.5 \* X \* sqr (t)**

Dist = id

‘=’ = operator

0.5 = number

‘\*’ = operator

X = id

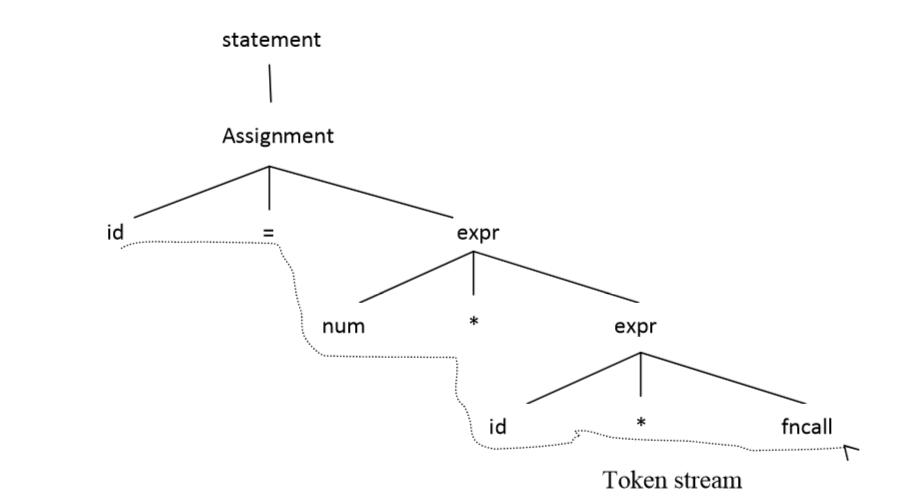
‘ \* ‘ = operator

sqr(t) = function call

**Syntax Analysis**

Syntax analysis is the second phase of the compiler. It is also known as parser. It receives the token stream as input from the lexical analyser of the compiler and generates syntax tree as the output.

Syntax tree: It is a tree in which interior nodes are operators and exterior nodes are operands. Example: For a=b+c\*2, syntax tree is



**Semantic Analysis**

Semantic analysis is the third phase of the compiler. It receives input from the syntax analysis as parse tree and checks whether the given syntax is correct or not. It also performs type conversion of all the data types into real data types.

The semantic routines of a compiler perform two main tasks: firstly, checking to make sure that each series of tokens will be understood by the computer when it is fully translated to machine code, and secondly, converting the series of tokens one step closer to machine code.

1. The first task takes a series of tokens, called a production, and checks it to see if it makes sense (or meaningful). For example, a production may be correct as far as the parser is concerned, but the semantic routines check whether the variables have been declared, and are of the right type.
2. If the production is meaningful, the semantic routine reduces the production for the next phase of compilation, called code generation. Most of the code for the compiler lies here in the semantic routines and thus takes up a majority of the compilation time.

**Implementation details**

The lex program identifies all the tokens present in the code and displays it with the line number.We have written the grammars for all the necessary identifiers and keywords.Keywords include all the predefined words whose meanings are defined.So all the keywords are present in the grammar of keywords.For identifier we have taken care that it should start with a alphabet then can be followed by integers or alphabets.For all types of operators grammars are specified.Also for decimal , float , integer , long long integer , scientific float we have specified the grammars which matches them.For comments we have  done this type of pattern

{START} ( {SIMPLE} | {COMPLEX} ) \* {END}  
The SIMPLE pattern should recognise anything legal, but that can't be confused with END, and only consists of a single character. It is usually of the form [...] or [^...]. The COMPLEX pattern is used to deal with the more difficult cases, possibly involving several input characters.Also grammar for header file is defined.

For identifying the string literal we have written   [a-zA-Z\_]?\"(\\.|[^\\"])\*"\"" which means it can start from any alphabets and should be within inverted commas (i.e “ “ ).

**Code for Scanner.l file**

%{

#include"scanner1.h"

#include"hashtable.c"

int line\_no=1;

int prev=0; char ptext[100]="NULL",ptype[100]="NULL";

%}

identifier [a-zA-Z][\_a-zA-Z0-9]\*

header "#include"[ ]\*"<"{identifier}".h"?">"

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"

digits [0-9]+

decimal       0|[1-9][0-9]\*

lint {decimal}"L"

llint {decimal}"LL"

double {decimal}?"."{digits}

float {double}"f"

scientific    {double}"e"{decimal}

scientificf  {scientific}"f"

str\_literal  [a-zA-Z\_]?\"(\\.|[^\\"])\*"\""

character     "'"."'"

p\_operator "->"|"/="|"%="|">>="|"="|"+="|"-="|"\*="

a\_operator "<<"|">>"|"=="|"<="|">="|"<<="|"&="|"^="|"|="|[-+\*%/<>&|^]

b\_operator    "&&"|"||"|"!="

i\_operator    "++"|"--"

space      [ \t]

next\_line     \n

array "["|"]"

comma ","

colon ":"

semicolon ";"

o\_brace "{"

c\_brace "}"

o\_paren       "("

c\_paren ")"

%x mlcomment

%x slcomment

%%

"/\*" BEGIN(mlcomment);

<mlcomment>[^\*\n]\* ;

<mlcomment>\n      ;

<mlcomment>"\*"+[^/] ;

<mlcomment>"\*"+"/" BEGIN(INITIAL);

"//" BEGIN(slcomment);

<slcomment>[^\n]\* ;

<slcomment>\n BEGIN(INITIAL);

{header} return HEADER;

{keyword} return KEYWORD;

{character} return CHAR;

{decimal} return DECIMAL;

{lint} return LONG\_INT;

{double} return DOUBLE;

{llint} return LONG\_LONG\_INT;

{float} return FLOAT;

{scientific} return SCIENTIFIC;

{scientificf} return SCIENTIFIC\_F;

{str\_literal} return STR\_LITERAL;

{identifier} return IDENTIFIER;

{p\_operator} return P\_OPERATOR;

{a\_operator} return A\_OPERATOR;

{b\_operator} return B\_OPERATOR;

{i\_operator} return I\_OPERATOR;

{array}   return ARRAY;

{comma}   return COMMA;

{colon}   return COLON;

{o\_brace} return O\_BRACE;

{c\_brace} return C\_BRACE;

{o\_paren} return O\_PAREN;

{c\_paren} return C\_PAREN;

{semicolon}   return SEMICOLON;

{next\_line} ++line\_no;

{space} ;

. {if(prev!=line\_no){printf("%s:Invalid character at line %d\n",yytext,line\_no);prev=line\_no;}}

%%

int main(void){

int ntoken=0;

yyin = fopen("abc.txt","r");

printf("Invalid:");

ntoken=yylex();

char type[100];

while(ntoken){

switch(ntoken){

case CHAR : strcpy(type,"CHARACTER");

break;

case HEADER : strcpy(type,"HEADER");

break;

case DATATYPE   : strcpy(type,"DATATYPE");

break;

case KEYWORD : strcpy(type,"KEYWORD");

  break;

case STR\_LITERAL : strcpy(type,"STRING");

break;

case IDENTIFIER : strcpy(type,"IDENTIFIER");

break;

case P\_OPERATOR : strcpy(type,"PRIMARY\_EXP\_OPERATOR");

break;

case A\_OPERATOR : strcpy(type,"ASSIGNMENT\_OPERATOR");

break;

case B\_OPERATOR : strcpy(type,"BINARY\_OPERATOR");

break;

case I\_OPERATOR : strcpy(type,"INCREMENT\_OPERATOR");

break;

case ARRAY : strcpy(type, “ARRAY-SUBSCRIPT OPERATOR");

break;

case COMMA : strcpy(type,"COMMA");

break;

case COLON : strcpy(type,"PUNCTUATOR");

break;

case O\_BRACE    : strcpy(type,"OPENIN'BRACE");

break;

case C\_BRACE : strcpy(type,"CLOSIN'BRACE");

break;

case O\_PAREN     : strcpy(type,"OPENIN'PARENTHESES");

break;

case C\_PAREN : strcpy(type,"CLOSIN'PARENTHESES");

break;

case DECIMAL    : strcpy(type,"DECIMAL");

break;

case LONG\_INT   :    strcpy(type, “LONG INT");

break;

case LONG\_LONG\_INT: strcpy(type, “LONG LONG INT");

break;

case FLOAT      :    strcpy(type, “FLOAT");

break;

case DOUBLE    : strcpy(type,"DOUBLE");

break;

case SCIENTIFIC :    strcpy(type, “SCIENTIFIC");

break;

case SCIENTIFIC\_F :   strcpy(type,"SCIENTIFIC\_F");

break;

case SEMICOLON   : strcpy(type, “SEMICOLON");

break;

}

char pptext[100]="NULL";

strcpy(pptext,ptext);

insert(yytext,type,line\_no);

strcpy(ptext,yytext);

strcpy(ptype,type);

ntoken = yylex();

if(!strcmp(yytext,"(")){

if(!strcmp(ptype,"IDENTIFIER")){

insertS(ptext,"function",line\_no);

}

}

else{

if(!strcmp(ptype,"IDENTIFIER")){

insertS(ptext,"identifier",line\_no);

}

}

}

printf("\n");

display();

displayS();

return 0;

} int yywrap(){ return 1;}

**Code for Scanner.h**

#define COLON 1  
#define SEMICOLON 2  
#define IDENTIFIER 3  
#define DECIMAL 4  
#define SCIENTIFIC 5  
#define STR\_LITERAL 6  
#define COMMA 7  
#define SCIENTIFIC\_F 8  
#define INT 10  
#define LONG\_INT 11  
#define LONG\_LONG\_INT 12 #define FLOAT 14  
#define DOUBLE 15  
#define LONG\_DOUBLE 16  
#define BOOL 17  
#define IF 18  
#define ELSE 19  
#define ELSE\_IF 20  
#define WHILE 21  
#define KEYWORD 22  
#define CHAR 23  
#define P\_OPERATOR 24  
#define U\_OPERATOR 25  
#define B\_OPERATOR 26  
#define T\_OPERATOR 27  
#define A\_OPERATOR 28  
#define ARRAY 29  
#define COMMENT 30  
#define ERR 31  
#define I\_OPERATOR 32  
#define HEADER 33  
#define DATATYPE 34  
#define O\_BRACE 35  
#define C\_BRACE 36  
#define FPC 37  
#define O\_PAREN 38  
#define C\_PAREN 39

**Code for hashtable.c**

#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
#include <stdbool.h>  
int size=1000;  
int key=0;float x = .1;  
struct token{  
   char name[100],type[100];  
   int attribute,line;  
};  
                              
struct token\* hash[1000];   
struct token\* item;  
  
void insert(char data[], char type[], int line) {  
   struct token \*item = (struct token\*) malloc(sizeof(struct token));  
   strcpy(item->name,data);   
   strcpy(item->type,type);   
   item->line = line;  
   hash[key++] = item;  
}  
  
void display(){  
   printf("\t\tToken-Name\t\t\t\tToken-Type\t\tLine no\n");  
   printf("\t--------------------------------------------------------------------------------\n");  
   for(int i=0;i<key;++i){  
      if(hash[i] != NULL)  
         printf("%30s\t\t%30s\t\t%d\n",hash[i]->name,hash[i]->type,hash[i]->line);  
   }  
}

**Test Cases :**

**#1 :**

|  |  |
| --- | --- |
|  | #include<stdio.h> |
|  | int main() |
|  | { |
|  | int a; int b; scanf("%d%d",&a,&b); |
|  | int product=a\*b; int sum=a+b; |
|  | if(a!=0){ |
|  | double divide = b/a; |
|  | } |
|  | return 0; } |
|  | **Expected Output** : This code should not show any errors . |
|  | https://lh5.googleusercontent.com/Z6OPyB4YmquSyzkXNvkAe1SdZPQG-D-E8OUo6aDAPU7LXvNiyIH7fq5DBjvl2ACuyer7hIhi4zT_QS5Ya47B0aCoBJc050w2bCb3ypO1k-4ueq9tSZSt8lnr-6fc4dEcDEFUEa3O |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **#2:** | #include<stdio.h>  void find\_max(long int x,long int y)  {  if ( a > b )  {  printf ("a is greater\n");  }  else  {  printf ("b is greater\n");  }  }  int main()  {  long int A[2];  A[1] = 2;  A[0] = 3;  find\_max(A[1],A[0]);  return 0;  }  **Expected output**: This code will not show errors and it should display the data types, keywords, identifiers, functions and operators as tokens. |
|  | https://lh4.googleusercontent.com/Q6gnAVP2Ry81aBLoFWCskz9iBLFlpdLmeWEeosemvFOhhDK7KfzLNeR3iQu5SSfqixwD7OL2SykyZdgwHEQComDcNVJOf10VL71iQDi74qHzEkNTLUrC-USmJe5Y7geX6GOhx1gW |
|  |  |
|  |  |
| **#3 :** | #include<stdio.h>  int main(void)  {  long long int i=0,j=5;  while (i <= 5)  {  i = i + 1;  }  do  {  j = j - 1;  }  while (j >= 0);  Return 0;  }  **Expected output** : This code contains no errors.  https://lh5.googleusercontent.com/h_6sTHtTFhLFN7PrjUOTW0aU9iQSL6Hf7jeoC_uDL3tmxyJJX0QS_X8yyQyIoz6oNh_jg7HO9JH9Klac-j9zjmn3G4NAKs2yr2UvQEu68Rb6RZJ_l6Zc98WblFMpxDMIwIm865ej |
|  |  |
|  |  |
| **#4:** |  |
|  |  |

#include<stdio.h>

int main()

{

”

// single comment

/\* multi line comment \*/

/\*/

}

**Expected output**: The output will be error because of comment syntax and invalid “ sign.

