COMPILER DESIGN LAB

PROJECT – 1

LEXICAL ANALYSER

- Tejas Rafaliya – 15CO250

- Siddhartha Chowdhuri – 15CO246

Table of Contents

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 supports 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.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

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 analyser represents these lexemes in the form of tokens as: <token name, attribute value>.

Sequences of characters in a token are called lexemes.

Functions of lexical analyser:

1. Tokenization i.e. dividing the program into valid tokens.

2. Remove white space characters.

3. Remove comments.

4. It also provides help in generating error message by providing row number and column number.

The lexical analyser identifies the error with the help of automation machine and the grammar of the given language on which it is based like C,C++.

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 analyser keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyser 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.

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:

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.

Lex code for the lexical analyser:

```
%
{
    #include <stdio.h>
    #include <string.h>
    #include <math.h>
    #include <stdlib.h>
    #include <stdbool.h>
    struct symboltable {
       char token[100];
       char type[100];
    };
    struct symboltable* stable[10000];
    struct symboltable* ctable[10000];
    struct symboltable* dummyItem;
    //strcpy (dummyItem->token,"-1");
    struct symboltable* item;
    int hashCode(char token[100]) {
       int i,x=0,j=strlen(token);
       x=0;
       for(i=j;i>=0;i--)
       {
            x=x*10+token[i];
            x%=10000;
       }

       return x%10000;
    }
    struct symboltable *search(char token[100]) {

       char hashIndex = hashCode(token);
       while(stable[hashIndex] != NULL) {

          if(stable[hashIndex]->token == token)
             return stable[hashIndex];
          ++hashIndex;
          hashIndex%=10000;

       }

       return NULL;
    }
    void insert(char a[100],char b[100]) {
       printf("%s - %s\n",a,b);
```

```
   struct symboltable *item = (struct symboltable*) malloc(sizeof(struct
symboltable));
   strcpy(item->type,b);
   strcpy(item->token,a);
   char token[100];
   strcpy(token,item->token);
   int hashIndex = hashCode(token);
   if (stable[hashIndex]!=NULL)
        if (strcmp(stable[hashIndex]->token,a)==0)
        return;
   while(stable[hashIndex] != NULL && stable[hashIndex]->token != "-1") {

      ++hashIndex;
      hashIndex%=10000;
   }

   stable[hashIndex] = item;
}
void cinsert(char a[100],char b[100]) {
   printf("%s - %s\n",a,b);
   struct symboltable *item = (struct symboltable*) malloc(sizeof(struct
symboltable));
   strcpy(item->type,b);
   strcpy(item->token,a);
   char token[100];
   strcpy(token,item->token);
   int hashIndex = hashCode(token);
   if (ctable[hashIndex]!=NULL)
        if (strcmp(ctable[hashIndex]->token,a)==0)
        return;
   while(ctable[hashIndex] != NULL && ctable[hashIndex]->token != "-1") {

      ++hashIndex;
      hashIndex%=10000;
   }

   ctable[hashIndex] = item;
}
struct symboltable* delete(struct symboltable* item) {
   char token[100];
   strcpy(token,item->token);
   int hashIndex = hashCode(token);
   while(stable[hashIndex] != NULL) {
      if(stable[hashIndex]->token == token) {
         struct symboltable* temp = stable[hashIndex];
         strcpy(stable[hashIndex]->token,"-1");
```

```
         return temp;
      }

      //go to next cell
      ++hashIndex;
               hashIndex%=10000;
   }

   return NULL;
}
int i=0;
%}
letter [a-zA-Z]
digit[0-9]
%%
{digit}+("E"("+"|"-")?{digit}+)? {cinsert(yytext,"Real Number");}
{digit}+"."{digit}+("E"("+"|"-")?{digit}+)? {cinsert(yytext,"Floating point number");}
"\""(.*)"\"" {cinsert(yytext,"Character constant");}
"#include <"{letter}*".h>"|"#include<"{letter}*".h>" {insert(yytext,"Include
statement"); }
"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"|"whi
le" {insert(yytext,"Keyword");}
({letter}|"_")({letter}|{digit})* {insert(yytext,"Identifier");}
"&&"|"<"|">"|"<="|">="|"="|"+"|"-"|"?"|"*"|"/"|"%"|"&"|"||"
{insert(yytext,"Operators");}
"{"|"["|"(" {insert(yytext,"Opening bracket"); }
"}"|"]"|")" {insert(yytext,"Closing bracket");}
"#"|"'"|"."|"\""|"," {insert(yytext,"Special characters");}
"\;" {insert(yytext,"Delimiter");}
"%d"|"%s"|"%c"|"%f"|"%e" {insert(yytext,"Format Specifier");}
"\\n" {insert(yytext,"New line");}
\/\/.*"\n" {insert(yytext,"Single line Commment");}
"/*"([^*]|\*+[^*/])*\*+"/" {insert(yytext,"Multi Line Commment");}
"@"|"$" {insert(yytext,"Lexical error");}
" "|"\t"|"\n"
%%
void display() {
   int i = 0;

   for(i = 0; i<10000; i++) {

      if(stable[i] != NULL && stable[i]->token!="-1")
         printf("%s - %s\n",stable[i]->token,stable[i]->type);
   }
```

```
}
void cdisplay() {
   int i = 0;

   for(i = 0; i<10000; i++) {

      if(ctable[i] != NULL && ctable[i]->token!="-1")
         printf("%s - %s\n",ctable[i]->token,ctable[i]->type);
   }
}
 int yywrap()
 {
 return 1;
 }
 int main()
 {
 yyin=fopen("abc.txt","r");
 yylex();
 int j;
 printf("\n\nSymbol table: \n \n");
 display();
 printf("\n\nConstant table: \n \n");
 cdisplay();
 return 0;
 }
```

Screenshots of Outputs:

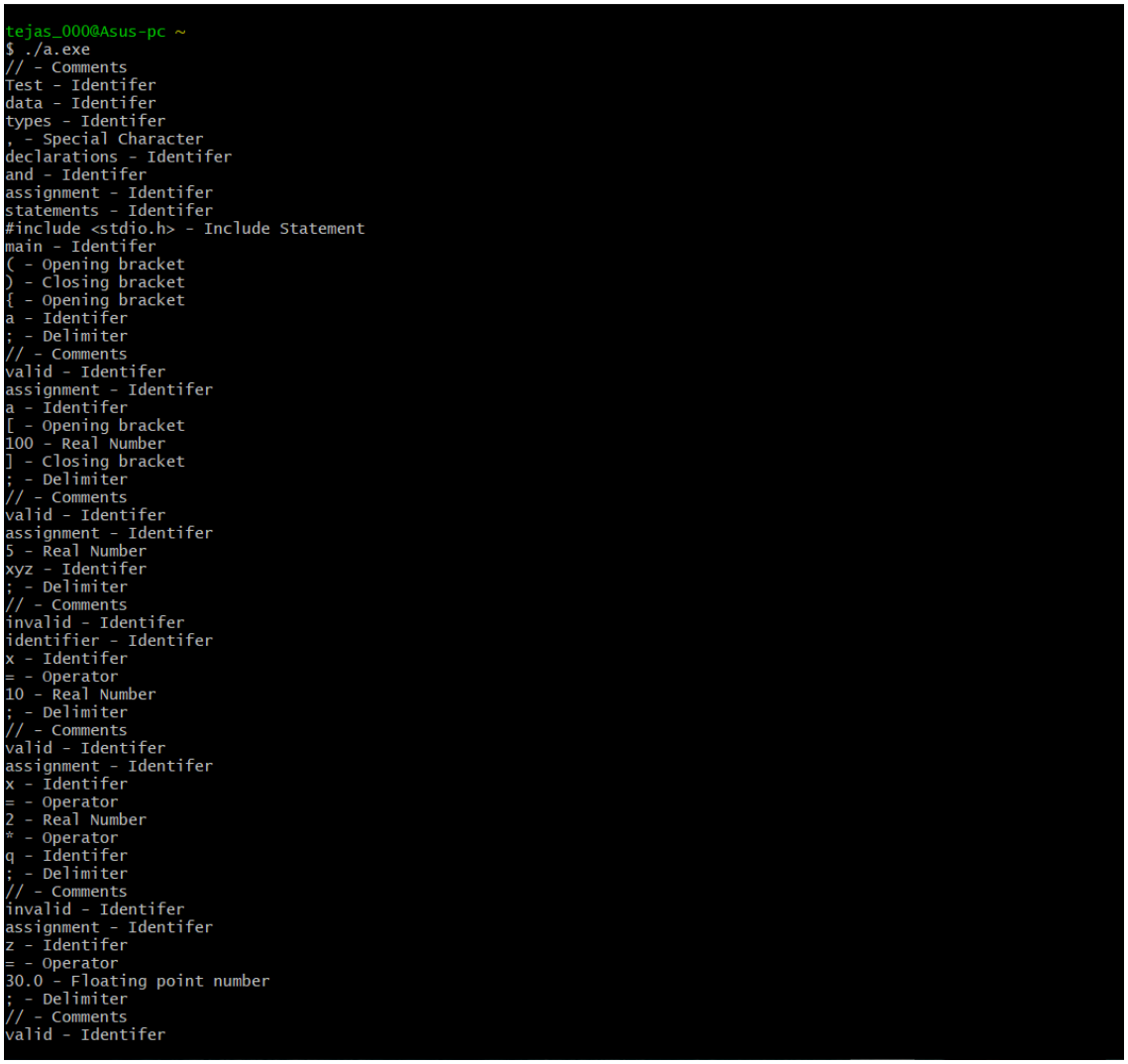
1. **Test case for datatypes:**

   Input:

```
// Test case for data types, declarations and assignment statements

#include <stdio.h>

int main()
{
        char a;             //valid char assignment
        char a[100];        //valid char assignment

        int 5xyz;           //invalid identifier
        int x=10;           //valid int assignment
         x=2*q;              //invalid assignment

         float z=30.0;       //valid float assignment
         float @y=25.0;      //invalid float assignment
        float y=25.1.0;      //invalid float assignment
```

```
        z=x+y;              //valid statement
       return 0;

}
```

Output:



```
assignment - Identifer
@ - Lex error Invalid token
y - Identifer
= - Operator
25.0 - Floating point number
; - Delimiter
// - Comments
invalid - Identifer
assignment - Identifer
y - Identifer
= - Operator
25.1 - Floating point number
. - Special Character
0 - Real Number
; - Delimiter
// - Comments
invalid - Identifer
assignment - Identifer
z - Identifer
= - Operator
x - Identifer
+ - Operator
y - Identifer
; - Delimiter
// - Comments
valid - Identifer
statement - Identifer
0 - Real Number
; - Delimiter
} - Closing bracket
```

```
Symbol table:

( - Opening bracket
) - Closing bracket
* - Operators
+ - Operators
. - Special characters
; - Delimiter
= - Operators
@ - Lexical error
[ - Opening bracket
] - Closing bracket
a - Identifier
q - Identifier
x - Identifier
y - Identifier
z - Identifier
{ - Opening bracket
} - Closing bracket
//invalid identifier
 - Single line Commment
//invalid assignment
 - Single line Commment
//invalid float assignment
 - Single line Commment
//invalid float assignment
 - Single line Commment
#include <stdio.h> - Include statement
main - Identifier
int - Keyword
xyz - Identifier
char - Keyword
// Test case for data types, declarations and assignment statements
 - Single line Commment
float - Keyword
//valid char assignment
 - Single line Commment
//valid int assignment
 - Single line Commment
//valid float assignment
 - Single line Commment
//valid statement
 - Single line Commment
return - Keyword


Constant table:

0 - Real Number
2 - Real Number
5 - Real Number
10 - Real Number
30.0 - Floating point number
25.0 - Floating point number
25.1 - Floating point number
100 - Real Number
```

2. Test case for loops and if-conditions:

Input:

```
// Test case for for loop, while loop, nested while loop and if-else statements

#include <stdio.h>

int main()
{
	int i,j=0;

	 for (i=0;i<10;i++)                    // valid for loop
	{
		printf("%d \n",i);
	}

	 for [i=0;i<10;i++]                     // invalid for loop
	 {
		printf("%d \n",i);
	 }

	 while (i<20)                           // valid nested while loops
	{
		j=0;
		while (j<3)
		{
			j++;
		}
	}
	while {x<9}                              // invalid while statement
		x++;

	 if (i<20)                              // valid if-else statements
	{
		printf("Hello World \n");
	}
	else if (i>20)
	{
		printf("Bye World \n");
	}
	else
		printf("Equality \n");


	if [i<20]                                 // invalid if-else statements
		printf("Nothing happens \n");

}
```

Output:

```
tejas_000@Asus-pc ~
$ ./a.exe
// - Comments
Test - Identifer
loop - Identifer
, - Special Character
loop - Identifer
, - Special Character
nested - Identifer
loop - Identifer
and - Identifer
- - Operator
statements - Identifer
#include <stdio.h> - Include Statement
main - Identifer
( - Opening bracket
) - Closing bracket
{ - Opening bracket
i - Identifer
, - Special Character
j - Identifer
= - Operator
0 - Real Number
; - Delimiter
( - Opening bracket
i - Identifer
= - Operator
0 - Real Number
; - Delimiter
i - Identifer
< - Operator
10 - Real Number
; - Delimiter
i - Identifer
+ - Operator
+ - Operator
) - Closing bracket
// - Comments
valid - Identifer
loop - Identifer
{ - Opening bracket
printf - Identifer
( - Opening bracket
" - Special Character
%d - Format specifier
\n - New line
" - Special Character
, - Special Character
i - Identifer
) - Closing bracket
; - Delimiter
} - Closing bracket
[ - Opening bracket
i - Identifer
= - Operator
0 - Real Number
; - Delimiter
i - Identifer
i - Identifer
< - Operator
10 - Real Number
; - Delimiter
i - Identifer
+ - Operator
+ - Operator
] - Closing bracket
// - Comments
invalid - Identifer
loop - Identifer
{ - Opening bracket
printf - Identifer
( - Opening bracket
" - Special Character
%d - Format specifier
\n - New line
" - Special Character
, - Special Character
i - Identifer
) - Closing bracket
; - Delimiter
} - Closing bracket
( - Opening bracket
i - Identifer
< - Operator
20 - Real Number
) - Closing bracket
// - Comments
valid - Identifer
nested - Identifer
loops - Identifer
{ - Opening bracket
j - Identifer
= - Operator
0 - Real Number
; - Delimiter
( - Opening bracket
j - Identifer
< - Operator
3 - Real Number
) - Closing bracket
{ - Opening bracket
j - Identifer
+ - Operator
+ - Operator
; - Delimiter
} - Closing bracket
} - Closing bracket
{ - Opening bracket
x - Identifer
< - Operator
9 - Real Number
} - Closing bracket
// - Comments
invalid - Identifer
statement - Identifer
x - Identifer
```

```
+ - Operator
+ - Operator
; - Delimiter
( - Opening bracket
i - Identifer
< - Operator
20 - Real Number
) - Closing bracket
// - Comments
valid - Identifer
- - Operator
statements - Identifer
{ - Opening bracket
printf - Identifer
( - Opening bracket
" - Special Character
Hello - Identifer
World - Identifer
\n - New line
" - Special Character
) - Closing bracket
; - Delimiter
} - Closing bracket
( - Opening bracket
i - Identifer
> - Operator
20 - Real Number
) - Closing bracket
{ - Opening bracket
printf - Identifer
( - Opening bracket
" - Special Character
Bye - Identifer
World - Identifer
\n - New line
" - Special Character
) - Closing bracket
; - Delimiter
} - Closing bracket
printf - Identifer
( - Opening bracket
" - Special Character
Equality - Identifer
\n - New line
" - Special Character
) - Closing bracket
; - Delimiter
[ - Opening bracket
i - Identifer
< - Operator
20 - Real Number
] - Closing bracket
// - Comments
invalid - Identifer
- - Operator
statements - Identifer
printf - Identifer
```

```
Symbol table:

( - Opening bracket
) - Closing bracket
+ - Operators
, - Special characters
; - Delimiter
< - Operators
= - Operators
> - Operators
[ - Opening bracket
] - Closing bracket
i - Identifier
j - Identifier
x - Identifier
{ - Opening bracket
} - Closing bracket
#include <stdio.h> - Include statement
if - Keyword
main - Identifier
// valid for loop
 - Single line Commment
// valid nested while loops
 - Single line Commment
// valid if-else statements
 - Single line Commment
printf - Identifier
for - Keyword
int - Keyword
else - Keyword
// Test case for for loop, while loop, nested while loop and if-else statements
 - Single line Commment
// invalid for loop
 - Single line Commment
// invalid while statement
 - Single line Commment
// invalid if-else statements
 - Single line Commment
while - Keyword


Constant table:

0 - Real Number
3 - Real Number
9 - Real Number
10 - Real Number
20 - Real Number
"%d \n" - Character constant
"Bye World \n" - Character constant
"Nothing happens \n" - Character constant
"Hello World \n" - Character constant
"Equality \n" - Character constant
```

3. Test case for datatypes:

Input:

```
// Test case for defining user-defined functions and invoking them in the main
function

#include <stdio.h>

void printpattern()                                  //correct function declaration
{
       printf("*    \n");
}


int ret10[]                                         //incorrect function declaration
{
       return 10;
}

void checkg10(int x)                                 //correct function declaration
{
       if (x>10)
       {
              printf("%d is greater than 10\n",x);
       }
       else
              printf("%d is not greater than 10\n",x);
}
int main()
{
       int i,j;
        printpattern();                              //correct function call

       i=ret10{};                                   //incorrect function call

       checkg10(i);                                 //correct function call

}
```

Output:

```
tejas_DOO@Asus-pc ~
$ ./a.exe
// - Comments
Test - Identifer
defining - Identifer
user - Identifer
- - Operator
defined - Identifer
functions - Identifer
and - Identifer
invoking - Identifer
them - Identifer
in - Identifer
the - Identifer
main - Identifer
function - Identifer
#include <stdio.h> - Include Statement
printpattern - Identifer
( - Opening bracket
) - Closing bracket
// - Comments
correct - Identifer
function - Identifer
declaration - Identifer
{ - Opening bracket
printf - Identifer
( - Opening bracket
" - Special Character
* - Operator
\n - New line
" - Special Character
) - Closing bracket
; - Delimiter
} - Closing bracket
ret10 - Identifer
[ - Opening bracket
] - Closing bracket
// - Comments
incorrect - Identifer
function - Identifer
declaration - Identifer
{ - Opening bracket
10 - Real Number
; - Delimiter
} - Closing bracket
checkg10 - Identifer
( - Opening bracket
x - Identifer
) - Closing bracket
// - Comments
correct - Identifer
function - Identifer
declaration - Identifer
{ - Opening bracket
( - Opening bracket
x - Identifer
> - Operator
10 - Real Number
```

```
10 - Real Number
) - Closing bracket
{ - Opening bracket
printf - Identifer
( - Opening bracket
" - Special Character
%d - Format specifier
is - Identifer
greater - Identifer
than - Identifer
10 - Real Number
\n - New line
" - Special Character
, - Special Character
x - Identifer
) - Closing bracket
; - Delimiter
j - Closing bracket
printf - Identifer
( - Opening bracket
" - Special Character
%d - Format specifier
is - Identifer
not - Identifer
greater - Identifer
than - Identifer
10 - Real Number
\n - New line
" - Special Character
, - Special Character
x - Identifer
) - Closing bracket
; - Delimiter
} - Closing bracket
main - Identifer
( - Opening bracket
) - Closing bracket
{ - Opening bracket
i - Identifer
, - Special Character
j - Identifer
; - Delimiter
printpattern - Identifer
( - Opening bracket
) - Closing bracket
; - Delimiter
// - Comments
correct - Identifer
function - Identifer
call - Identifer
i - Identifer
= - Operator
ret10 - Identifer
{ - Opening bracket
} - Closing bracket
; - Delimiter
// - Comments
incorrect - Identifer
```

```
incorrect - Identifer
function - Identifer
call - Identifer
checkg10 - Identifer
( - Opening bracket
i - Identifer
) - Closing bracket
; - Delimiter
// - Comments
correct - Identifer
function - Identifer
call - Identifer
} - Closing bracket
```

```
Symbol table:

( - Opening bracket
) - Closing bracket
, - Special characters
; - Delimiter
= - Operators
> - Operators
[ - Opening bracket
] - Closing bracket
i - Identifier
j - Identifier
x - Identifier
{ - Opening bracket
} - Closing bracket
checkg10 - Identifier
//incorrect function declaration
 - Single line Commment
//incorrect function call
 - Single line Commment
#include <stdio.h> - Include statement
if - Keyword
//correct function declaration
 - Single line Commment
//correct function call
 - Single line Commment
//correct function call
 - Single line Commment
main - Identifier
ret10 - Identifier
void - Keyword
printpattern - Identifier
printf - Identifier
printf - Identifier
printf - Identifier
int - Keyword
else - Keyword
// Test case for defining user-defined functions and invoking them in the main  function
 - Single line Commment
return - Keyword


Constant table:

10 - Real Number
"%d is greater than 10\n" - Character constant
"%d is not greater than 10\n" - Character constant
"*    \n" - Character constant
```

4. Test case for comments:

Input:

```
#include <stdio.h>

int main()
{

//This is a commment                                  Valid comment

/*
This is also a comment                                Valid comment
*/

                                                    //Invalid comment
/*
This comment has no end
}
```

Output:

```
tejas_000@Asus-pc ~
$ ./a.exe
#include <stdio.h> - Include Statement
main - Identifer
( - Opening bracket
) - Closing bracket
{ - Opening bracket
// - Comments
This - Identifer
is - Identifer
a - Identifer
commment - Identifer
Valid - Identifer
comment - Identifer
/* - Comments
This - Identifer
is - Identifer
also - Identifer
a - Identifer
comment - Identifer
Valid - Identifer
comment - Identifer
*/ - Comments
// - Comments
Invalid - Identifer
comment - Identifer
/* - Comments
This - Identifer
comment - Identifer
has - Identifer
no - Identifer
end - Identifer
} - Closing bracket
```

```
Symbol table:

( - Opening bracket
) - Closing bracket
* - Operators
/ - Operators
{ - Opening bracket
} - Closing bracket
#include <stdio.h> - Include statement
comment - Identifier
end - Identifier
no - Identifier
main - Identifier
has - Identifier
int - Keyword
//This is a commment                              Valid comment
 - Single line Commment
This - Identifier
/*
This is also a comment                            Valid comment
*/ - Multi Line Commment
//Invalid comment
 - Single line Commment


Constant table:
```