Intermediate Code Generator for the C language

National Institute of Technology Karnataka Surathkal

Date: 7th November 2020

Submitted To:
Prof. P. Santhi Thilagam
CSE Dept, NITK

Group Members:
Niranjan S Y 181CO136
Rajath C Aralikatti 181CO241
Shruthan R 181CO250
Varun N R 181CO134

Abstract

This report contains details of the tasks finished as a part of the Intermediate Code Generation Phase of the Compiler Design Lab. The tasks involve making an intermediate code generator for the C language from the annotated parse tree built in the Semantic Phase. The intermediate code produced is in the three address code form which is a type of machine independent linear representation. We provide explanations for the developed code, along with running test cases to demonstrate its working.

Objective

The objective of this phase is to generate a three address code for the C language using LEX and YACC scripts.

Functions

The proposed implementation for the mini C compiler with three address code generation supports the following features:

- Generation of three address code
- Supports for int, char data types
- Support for nested for loops and while loops
- Support for if-else and nested if-else
- Support for operations such as addition, multiplication, division and modulo
- Support for single and multi line comments
- Construction of symbol and constant table
- Semantic checking with the production of appropriate error messages

Requirements

- GCC, the GNU Compiler Collection
- FLEX (Fast Lexical Analyser Generator)
- YACC (Yet Another Compiler-Compiler)

TABLE OF CONTENTS

1. Introduction

1.1 Intermediate Code Generation:

Compilers serve as translators between the high-level programming language that humans use, and the machine code that is actually run on computers. The purpose of intermediate code generation is to connect the front-end (language specific) with the backend (hardware specific). The reason intermediate code is produced is to make compilers more platform independent and to avoid rewriting backends for different language-hardware combinations. The frontend first converts the source program into intermediate code. The backend, which is specific to the hardware (x86, ARM, etc) then converts the intermediate code to machine code. For this reason the Intermediate Code Generation Phase can be thought of as the glue that binds the frontend and the backend of the compiler design stages.

The following are commonly used intermediate code representation:

- Abstract Syntax Tree
- Three Address Code
- Directed Acyclic Graph
- Postfix Notation

Three Address Code

A statement involving no more than three references (two for operands and one for result) is known as a three address statement. A sequence of three address statements is known as three address code. It is a form of tuple-based intermediate code which is easy to generate and convert to machine code.

The general representation is in the form t1 = t2 op t3. It makes use of at most three addresses (there can be less than three) and one operator to represent an expression and the value computed at each instruction is stored in a temporary variable generated by the compiler. The compiler decides the order of operation given by the three address code.

Example: The three address code representation for the expression (x + y) * (y + z) + (x + y + z)

```
t1 = x + y
t2 = y + z
t3 = t1 * t2
t4 = t1 + z
t5 = t3 + t4
```

Yacc Script

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

The structure of our Yacc script is given below. Files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section
%%
Rules section
%%
C code
section

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file. The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code. The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

1.2 Execution Steps

The workflow is explained as under:

- Compile the script using Yacc tool
  $ yacc -d parser.y
- Compile the flex *s*cript using Flex tool
  $ lex scanner.l
- After compiling the lex file, lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
- Compilation of these scripts is done with the options –ll, –ly and -w
- The executable file is generated, which on running parses the C file given as a command line input
  $ ./a.out test.c

2. Design of Programs

2.1 Lexer Code

scanner.l

```
%{
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
   #include "y.tab.h"
   #include "symbolTable.h"
   #include "utils.h"


   char curid[20];
   char curtype[20];
   char curval[20];

%}

DE "define"
IN "include"

%%
\n  {yylineno++;}
([#]["   "]*({IN})[   ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|"
"|"\t"]  { }
```

```
([#]["    "]*({DE})["    "]*([A-Za-z]+)("    ")*[0-9]+)/["\n"|\/|"    "|"\t"]
{ }
\/\/(.*)
{ }
\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/
{ }
[ \n\t] ;
";"             { return(';'); }
","             { return(','); }
("{")           { scope_start(); return('{'); }
("}")           { scope_end(); return('}'); }
"("             { return('('); }
")"             { return(')'); }
("["|"<:")      { return('['); }
("]"|":>")      { return(']'); }
":"             { return(':'); }
"."             { return('.'); }

"char"                          {   strcpy(curtype,yytext);   insertST(yytext,
"Keyword");return CHAR;}
"double"            {  strcpy(curtype,yytext); insertST(yytext,  "Keyword");
return DOUBLE;}
"else"          { insertST(yytext, "Keyword"); return ELSE;}
"float"             {  strcpy(curtype,yytext); insertST(yytext,  "Keyword");
return FLOAT;}
"while"         { insertST(yytext, "Keyword"); return WHILE;}
"do"            { insertST(yytext, "Keyword"); return DO;}
"for"           { insertST(yytext, "Keyword"); return FOR;}
"if"            { insertST(yytext, "Keyword"); return IF;}
"int"               {  strcpy(curtype,yytext); insertST(yytext,  "Keyword");
return INT;}
"long"              {  strcpy(curtype,yytext); insertST(yytext,  "Keyword");
return LONG;}
"return"        { insertST(yytext, "Keyword"); return RETURN;}
"short"             {  strcpy(curtype,yytext); insertST(yytext,  "Keyword");
return SHORT;}
"signed"            {  strcpy(curtype,yytext); insertST(yytext,  "Keyword");
return SIGNED;}
"sizeof"        { insertST(yytext, "Keyword"); return SIZEOF;}
"struct"          {  strcpy(curtype,yytext);    insertST(yytext,  "Keyword");
return STRUCT;}
"unsigned"      { insertST(yytext, "Keyword");   return UNSIGNED;}
```

```
"void"             { strcpy(curtype,yytext);   insertST(yytext, "Keyword");
return VOID;}
"break"          { insertST(yytext, "Keyword");  return BREAK;}



"++"             { return increment_operator; }
"--"             { return decrement_operator; }
"<<"             { return leftshift_operator; }
">>"             { return rightshift_operator; }
"<="             { return lessthan_assignment_operator; }
"<"              { return lessthan_operator; }
">="             { return greaterthan_assignment_operator; }
">"              { return greaterthan_operator; }
"=="             { return equality_operator; }
"!="             { return inequality_operator; }
"&&"             { return AND_operator; }
"||"             { return OR_operator; }
"^"              { return caret_operator; }
"*="             { return multiplication_assignment_operator; }
"/="             { return division_assignment_operator; }
"%="             { return modulo_assignment_operator; }
"+="             { return addition_assignment_operator; }
"-="             { return subtraction_assignment_operator; }
"<<="            { return leftshift_assignment_operator; }
">>="            { return rightshift_assignment_operator; }
"&="             { return AND_assignment_operator; }
"^="             { return XOR_assignment_operator; }
"|="             { return OR_assignment_operator; }
"&"              { return amp_operator; }
"!"              { return exclamation_operator; }
"~"              { return tilde_operator; }
"-"              { return subtract_operator; }
"+"              { return add_operator; }
"*"              { return multiplication_operator; }
"/"              { return division_operator; }
"%"              { return modulo_operator; }
"|"              { return pipe_operator; }
\=               { return assignment_operator;}

\"[^\n]*\"/[;|,|\)]                                    {strcpy(curval,yytext);
insertCT(yytext,"String Constant"); return string_constant;}
```

```
\'[A-Z|a-z]\'/[;|,|\)|:]                              {strcpy(curval,yytext);
insertCT(yytext,"Character Constant"); return character_constant;}
[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[     {strcpy(curid,yytext);  insertST(yytext,
"Array");  return array_identifier;}
[1-9][0-9]*|0/[;|,|"  "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
{strcpy(curval,yytext);  insertCT(yytext,  "Number  Constant");  yylval  =
atoi(yytext); return integer_constant;}
([0-9]*)\.([0-9]+)/[;|,|"     "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
{strcpy(curval,yytext);  insertCT(yytext,  "Floating  Constant");  return
float_constant;}
[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext); insertST(curid,"Identifier");
return identifier;}

(.?) {
       if(yytext[0]=='#')
       {
                    printf("Error  in  Pre-Processor  directive  at  line  no.
%d\n",yylineno);
       }
       else if(yytext[0]=='/')
       {
           printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);
       }
       else if(yytext[0]=='"')
       {
           printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);
       }
       else
       {
           printf("ERROR at line no. %d\n",yylineno);
       }
       printf("%s\n", yytext);
       return 0;
}

%%
```

2.2 Parser Code

parser.y
```
%{
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>


   void yyerror(char* s);
   int yylex();
   void ins();
   void insV();
   int flag=0;

   extern char curid[20];
   extern char curtype[20];
   extern char curval[20];
   extern int currnest;
   void deletedata (int );
   char currfunctype[100];
   char currfunc[100];
   char currfunccall[100];

   int params_count=0;
   extern int call_params_count=0;

%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
%token ENDIF
%expect 1

%token identifier array_identifier func_identifier
%token integer_constant string_constant float_constant character_constant

%nonassoc ELSE
```

```
%right leftshift_assignment_operator rightshift_assignment_operator
%right XOR_assignment_operator OR_assignment_operator
%right AND_assignment_operator modulo_assignment_operator
%right multiplication_assignment_operator division_assignment_operator
%right addition_assignment_operator subtraction_assignment_operator
%right assignment_operator

%left OR_operator
%left AND_operator
%left pipe_operator
%left caret_operator
%left amp_operator
%left equality_operator inequality_operator
%left            lessthan_assignment_operator            lessthan_operator
greaterthan_assignment_operator greaterthan_operator
%left leftshift_operator rightshift_operator
%left add_operator subtract_operator
%left multiplication_operator division_operator modulo_operator

%right SIZEOF
%right tilde_operator exclamation_operator
%left increment_operator decrement_operator


%start program

%%
program
           : declaration_list;

declaration_list
           : declaration D

D
           : declaration_list
           | ;

declaration
           : variable_declaration
           | function_declaration

variable_declaration
           : type_specifier variable_declaration_list ';'
```

```
variable_declaration_list
            : variable_declaration_list ',' variable_declaration_identifier
| variable_declaration_identifier;

variable_declaration_identifier
                                                            :     identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,cur
rnest); ins();  } vdi
                                                       |    array_identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,cur
rnest); ins();  } vdi;



vdi : identifier_array_type | assignment_operator simple_expression  ;

identifier_array_type
           : '[' initilization_params
           | ;

initilization_params
            : integer_constant ']' initilization {if($$ < 1) {printf("Wrong
array size\n"); exit(0);} }
           | ']' string_initilization;

initilization
           : string_initilization
           | array_initialization
           | ;

type_specifier
           : INT | CHAR | FLOAT  | DOUBLE
           | LONG long_grammar
           | SHORT short_grammar
           | UNSIGNED unsigned_grammar
           | SIGNED signed_grammar
           | VOID  ;

unsigned_grammar
           : INT | LONG long_grammar | SHORT short_grammar | ;

signed_grammar
```

```
            : INT | LONG long_grammar | SHORT short_grammar | ;

long_grammar
            : INT  | ;

short_grammar
            : INT | ;

function_declaration
                                    :    function_declaration_type
function_declaration_param_statement;

function_declaration_type
                  : type_specifier identifier '('  { strcpy(currfunctype,
curtype);       strcpy(currfunc,       curid);       check_duplicate(curid);
insertSTF(curid); ins(); };

function_declaration_param_statement
                     : {params_count=0;}params ')' {funcgen();} statement
{funcgenend();};

params
            : parameters_list { insertSTparamscount(currfunc, params_count);
}| { insertSTparamscount(currfunc, params_count); };

parameters_list
                            :   type_specifier   {   check_params(curtype);}
parameters_identifier_list ;

parameters_identifier_list
            : param_identifier parameters_identifier_list_breakup;

parameters_identifier_list_breakup
            : ',' parameters_list
            | ;

param_identifier
               : identifier { ins();insertSTnest(curid,1); params_count++; }
param_identifier_breakup;

param_identifier_breakup
            : '[' ']'
            | ;
```

```
statement
           : expression_statment | compound_statement
           | conditional_statements | iterative_statements
           | return_statement | break_statement
           | variable_declaration;

compound_statement
                          :  {currnest++;}  '{'    statment_list    '}'
{deletedata(currnest);currnest--;}  ;

statment_list
           : statement statment_list
           | ;

expression_statment
           : expression ';'
           | ';' ;

conditional_statements
                                    :   IF   '('   simple_expression   ')'
{label1();if($3!=1){printf("Condition    checking    is    not    of    type
int\n");exit(0);}} statement {label2();}  conditional_statements_breakup;

conditional_statements_breakup
           : ELSE statement {label3();}
           | {label3();};

iterative_statements
                       :  WHILE  '('  {label4();}  simple_expression  ')'
{label1();if($4!=1){printf("Condition    checking    is    not    of    type
int\n");exit(0);}} statement {label5();}
               | FOR '(' expression ';' {label4();} simple_expression ';'
{label1();if($6!=1){printf("Condition    checking    is    not    of    type
int\n");exit(0);}} expression ')'statement {label5();}
                  |  {label4();}DO  statement  WHILE  '('  simple_expression
')'{label1();label5();if($6!=1){printf("Condition checking is not of type
int\n");exit(0);}} ';';
return_statement
           : RETURN ';' {if(strcmp(currfunctype,"void")) {printf("Returning
void of a non-void function\n"); exit(0);}}
           | RETURN expression ';' {   if(!strcmp(currfunctype, "void"))
                                       {
                                             yyerror("Function is void");
                                         }

                                                  if((currfunctype[0]=='i' ||
currfunctype[0]=='c') && $2!=1)
                                         {
                                             printf("Expression doesn't match
return type of function\n"); exit(0);
                                         }

                                     };

break_statement
            : BREAK ';' ;

string_initilization
            : assignment_operator string_constant {insV();} ;

array_initialization
            : assignment_operator '{' array_int_declarations '}';

array_int_declarations
            : integer_constant array_int_declarations_breakup;

array_int_declarations_breakup
            : ',' array_int_declarations
            | ;

expression
            : mutable assignment_operator {push("=");} expression   {

if($1==1 && $4==1)
                                                                      {
                                                                      $$=1;
                                                                      }
                                                                      else

{$$=-1; printf("Type mismatch\n"); exit(0);}

codeassign();
                                                                   }
            | mutable addition_assignment_operator {push("+=");}expression {
```

```
if($1==1 && $4==1)
                                                                                  $$=1;
                                                                                  else

{$$=-1; printf("Type mismatch\n"); exit(0);}

codeassign();
                                                                               }
                   |  mutable  subtraction_assignment_operator  {push("-=");}
expression  {

if($1==1 && $4==1)
                                                                                  $$=1;
                                                                                  else

{$$=-1; printf("Type mismatch\n"); exit(0);}

codeassign();
                                                                               }
                 | mutable  multiplication_assignment_operator  {push("*=");}
expression {

if($1==1 && $4==1)
                                                                                  $$=1;
                                                                                  else

{$$=-1; printf("Type mismatch\n"); exit(0);}

codeassign();
                                                                               }
              | mutable division_assignment_operator {push("/=");}expression
{

if($1==1 && $4==1)
                                                                                  $$=1;
                                                                                  else

{$$=-1; printf("Type mismatch\n"); exit(0);}
                                                                               }
               | mutable modulo_assignment_operator {push("%=");}expression
{
```

```
if($1==1 && $3==1)
                                                                                    $$=1;
                                                                                    else

{$$=-1; printf("Type mismatch\n"); exit(0);}

codeassign();
                                                                                  }
               | mutable increment_operator                                      {
push("++");if($1 == 1) $$=1; else $$=-1; genunary();}
                                        |    mutable    decrement_operator
{push("--");if($1 == 1) $$=1; else $$=-1;}
            | simple_expression {if($1 == 1) $$=1; else $$=-1;} ;


simple_expression
               : simple_expression OR_operator and_expression {push("||");}
{if($1 == 1 && $3==1) $$=1; else $$=-1; codegen();}
            | and_expression {if($1 == 1) $$=1; else $$=-1;};

and_expression
                             :   and_expression   AND_operator   {push("&&");}
unary_relation_expression    {if($1  ==  1  &&  $3==1)  $$=1;  else  $$=-1;
codegen();}
              |unary_relation_expression {if($1 == 1) $$=1; else $$=-1;} ;


unary_relation_expression
               : exclamation_operator {push("!");} unary_relation_expression
{if($2==1) $$=1; else $$=-1; codegen();}
            | regular_expression {if($1 == 1) $$=1; else $$=-1;} ;

regular_expression
             : regular_expression relational_operators sum_expression {if($1
== 1 && $3==1) $$=1; else $$=-1; codegen();}
              | sum_expression {if($1 == 1) $$=1; else $$=-1;} ;

relational_operators
                         :  greaterthan_assignment_operator  {push(">=");}  |
lessthan_assignment_operator     {push("<=");}     |     greaterthan_operator
{push(">");}|     lessthan_operator     {push("<");}|     equality_operator
{push("==");}| inequality_operator {push("!=");} ;
```

```
sum_expression
            : sum_expression sum_operators term  {if($1 == 1 && $3==1) $$=1;
else $$=-1; codegen();}
            | term {if($1 == 1) $$=1; else $$=-1;};

sum_operators
            : add_operator {push("+");}
            | subtract_operator {push("-");} ;

term
               : term MULOP factor {if($1 == 1 && $3==1) $$=1; else $$=-1;
codegen();}
            | factor {if($1 == 1) $$=1; else $$=-1;} ;

MULOP
                  : multiplication_operator {push("*");}| division_operator
{push("/");} | modulo_operator {push("%");} ;

factor
            : immutable {if($1 == 1) $$=1; else $$=-1;}
            | mutable {if($1 == 1) $$=1; else $$=-1;} ;

mutable
            : identifier {
                          push(curid);
                          if(check_id_is_func(curid))
                              {printf("Function name used as Identifier\n");
exit(8);}
                          if(!checkscope(curid))

{printf("%s\n",curid);printf("Undeclared\n");exit(0);}
                          if(!checkarray(curid))
                               {printf("%s\n",curid);printf("Array ID has no
subscript\n");exit(0);}
                              if(gettype(curid,0)=='i' || gettype(curid,1)==
'c')
                          $$ = 1;
                          else
                          $$ = -1;
                          }
```

```
                                                    |    array_identifier
{if(!checkscope(curid)){printf("%s\n",curid);printf("Undeclared\n");exit(0)
;}} '[' expression ']'
                                              {if(gettype(curid,0)=='i' ||
gettype(curid,1)== 'c')
                        $$ = 1;
                        else
                        $$ = -1;
                        };

immutable
        : '(' expression ')' {if($2==1) $$=1; else $$=-1;}
        | call {if($1==-1) $$=-1; else $$=1;}
        | constant {if($1==1) $$=1; else $$=-1;};

call
        : identifier '('{

                   if(!check_declaration(curid, "Function"))
                   { printf("Function not declared"); exit(0);}
                   insertSTF(curid);
                   strcpy(currfunccall,curid);
                   if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')
                   {
                   $$ = 1;
                   }
                   else
                   $$ = -1;
                   call_params_count=0;
                   }
                   arguments ')'
                   { if(strcmp(currfunccall,"printf"))
                     {

if(getSTparamscount(currfunccall)!=call_params_count)
                        {
                           yyerror("Number of arguments in function
call doesn't match number of parameters");
                           exit(8);
                        }
                     }
                     callgen();
                   };
```

```
arguments
           : arguments_list | ;

arguments_list
           : arguments_list ',' exp { call_params_count++; }
           | exp { call_params_count++; };

exp   :   identifier   {arggen(1);}    |   integer_constant   {arggen(2);}    |
string_constant    {arggen(3);}     |    float_constant    {arggen(4);}     |
character_constant {arggen(5);} ;

constant
           : integer_constant  {  insV(); codegencon(); $$=1; }
           | string_constant   {  insV(); codegencon();$$=-1;}
           | float_constant    {  insV(); codegencon();}
           | character_constant{  insV(); codegencon();$$=1; };

%%

extern FILE *yyin;
extern int yylineno;
extern char *yytext;


void callgen()
{
   printf("refparam result\n");
   push("result");
   printf("call %s, %d\n",currfunccall,call_params_count);
}



int main(int argc , char **argv)
{
   yyin = fopen(argv[1], "r");
   printf("\n----------------------------------------------------------\n");
   printf("              3 - ADDRESS CODE                           |\n");
   printf("----------------------------------------------------------\n\n");
   yyparse();
   printf("----------------------------------------------------------\n\n");
   if(flag == 0)
   {
       printf("Status: Parsing Complete\n\n");

printf("-------------------------------------------------------------------
---------------------------------\n");
         printf("                                              SYMBOL TABLE
|\n");

printf("-------------------------------------------------------------------
---------------------------------\n");
       printST();

       printf("\n\n\n----------------------------------------\n");
       printf("           CONSTANT TABLE              |\n");
       printf("----------------------------------------\n");
       printCT();
   }
}

void yyerror(char *s)
{
   printf("%d %s %s\n", yylineno, s, yytext);
   flag=1;
   printf("Status: Parsing Failed\n");
   exit(7);
}

void ins()
{
   insertSTtype(curid,curtype);
}

void insV()
{
   insertSTvalue(curid,curval);
}

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

2.3 Symbol Table

symbolTable.h
```
struct symboltable
{
   char name[100];
   char class[100];
   char type[100];
   char value[100];
   int nestval;
   int lineno;
   int length;
   int params_count;
   int scope;
}ST[1001];

struct constanttable
{
   char name[100];
   char type[100];
   int length;
}CT[1001];

int scope_stack[100] = {0};
int scope_index = 1;
int scope_val = 1;
int currnest = 0;
extern int yylval;

void scope_start()
{
   scope_stack[scope_index] = scope_val;
   scope_val++;
   scope_index++;
}

void scope_end()
{
   scope_index--;
   scope_stack[scope_index] = 0;
}

int hash(char *str)
```

```
{
   int value = 0;
   for(int i = 0 ; i < strlen(str) ; i++)
   {
       value = 10*value + (str[i] - 'A');
       value = value % 1001;
       while(value < 0)
           value = value + 1001;
   }
   return value;
}

int lookupST(char *str)
{
   int value = hash(str);
   if(ST[value].length == 0)
   {
       return 0;
   }
   else if(strcmp(ST[value].name,str)==0)
   {

       return value;
   }
   else
   {
       for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
       {
           if(strcmp(ST[i].name,str)==0)
           {

               return i;
           }
       }
       return 0;
   }
}

int lookupCT(char *str)
{
   int value = hash(str);
   if(CT[value].length == 0)
       return 0;
```

```
    else if(strcmp(CT[value].name,str)==0)
        return 1;
    else
    {
        for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            if(strcmp(CT[i].name,str)==0)
            {
                return 1;
            }
        }
        return 0;
    }
}

void insertSTline(char *str1, int line)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            ST[i].lineno = line;
        }
    }
}
void insertST(char *str1, char *str2)
{
    if(lookupST(str1))
    {
                    if(strcmp(ST[lookupST(str1)].class,"Identifier")==0   &&
strcmp(str2,"Array")==0)
        {
            printf("Error use of array\n");
            exit(0);
        }
        return;
    }
    else
    {
        int value = hash(str1);
        if(ST[value].length == 0)
        {
            strcpy(ST[value].name,str1);
            strcpy(ST[value].class,str2);
            ST[value].length = strlen(str1);
            ST[value].nestval = 9999;
            ST[value].params_count = -1;
            ST[value].scope = scope_stack[scope_index - 1];
            insertSTline(str1,yylineno);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            if(ST[i].length == 0)
            {
                pos = i;
                break;
            }
        }

        strcpy(ST[pos].name,str1);
        strcpy(ST[pos].class,str2);
        ST[pos].length = strlen(str1);
        ST[pos].nestval = 9999;
        ST[pos].params_count = -1;
        ST[pos].scope = scope_stack[scope_index - 1];
        insertSTline(str1,yylineno);
    }
}

void insertSTtype(char *str1, char *str2)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            strcpy(ST[i].type,str2);
        }
    }
}

void insertSTvalue(char *str1, char *str2)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0 && ST[i].nestval == currnest)
        {
            strcpy(ST[i].value,str2);
        }
    }
}


void insertSTnest(char *s, int nest)
{
    if(lookupST(s) && ST[lookupST(s)].nestval != 9999)
    {
            int pos = 0;
            int value = hash(s);
        for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            if(ST[i].length == 0)
            {
                pos = i;
                break;
            }
        }

        strcpy(ST[pos].name,s);
        strcpy(ST[pos].class,"Identifier");
        ST[pos].length = strlen(s);
        ST[pos].nestval = nest;
        ST[pos].params_count = -1;
        ST[pos].lineno = yylineno;
        ST[pos].scope = scope_stack[scope_index - 1];
    }
    else
    {
        for(int i = 0 ; i < 1001 ; i++)
        {
            if(strcmp(ST[i].name,s)==0 )
            {
                ST[i].nestval = nest;
            }
        }
    }
```

```
}

void insertSTparamscount(char *s, int count1)
{

   for(int i = 0 ; i < 1001 ; i++)
   {
       if(strcmp(ST[i].name,s)==0 )
       {
           ST[i].params_count = count1;
       }
   }
}

int getSTparamscount(char *s)
{
   for(int i = 0 ; i < 1001 ; i++)
   {
       if(strcmp(ST[i].name,s)==0 )
       {
           return ST[i].params_count;
       }
   }
   return -1;
}
void insertSTF(char *s)
{
   for(int i = 0 ; i < 1001 ; i++)
   {
       if(strcmp(ST[i].name,s)==0 )
       {
           strcpy(ST[i].class,"Function");
           return;
       }
   }

}

void insertCT(char *str1, char *str2)
{
   if(lookupCT(str1))
       return;
   else
```

```
    {
        int value = hash(str1);
        if(CT[value].length == 0)
        {
            strcpy(CT[value].name,str1);
            strcpy(CT[value].type,str2);
            CT[value].length = strlen(str1);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            if(CT[i].length == 0)
            {
                pos = i;
                break;
            }
        }

        strcpy(CT[pos].name,str1);
        strcpy(CT[pos].type,str2);
        CT[pos].length = strlen(str1);
    }
}

void deletedata (int nesting)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(ST[i].nestval == nesting)
        {
            ST[i].nestval = 99999;
        }
    }


}

int checkscope(char *s)
{
    int flag = 0;
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].name,s)==0)
        {
            if(ST[i].nestval > currnest)
            {
                flag = 1;
            }
            else
            {
                flag = 0;
                break;
            }
        }
    }
    if(!flag)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int check_id_is_func(char *s)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].name,s)==0)
        {
            if(strcmp(ST[i].class,"Function")==0)
                return 1;
        }
    }
    return 0;
}

int checkarray(char *s)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].name,s)==0)
        {
            if(strcmp(ST[i].class,"Array")==0)
            {
                return 0;
            }
        }
    }
    return 1;
}

int duplicate(char *s)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].name,s)==0)
        {
            if(ST[i].nestval == currnest)
            {
                return 1;
            }
        }
    }

    return 0;
}

int check_duplicate(char* str)
{
    for(int i=0; i<1001; i++)
    {
         if(strcmp(ST[i].name, str) == 0 && strcmp(ST[i].class, "Function")
== 0)
        {
            printf("Function redeclaration not allowed\n");
            exit(0);
        }
    }
}

int check_declaration(char* str, char *check_type)
{
    for(int i=0; i<1001; i++)
    {
         if(strcmp(ST[i].name, str) == 0 && strcmp(ST[i].class, "Function")
== 0 || strcmp(ST[i].name,"printf")==0 )
        {
            return 1;
        }
    }
    return 0;
}

int check_params(char* type_specifier)
{
    if(!strcmp(type_specifier, "void"))
    {
        printf("Parameters cannot be of type void\n");
        exit(0);
    }
    return 0;
}

char gettype(char *s, int flag)
{
        for(int i = 0 ; i < 1001 ; i++ )
        {
            if(strcmp(ST[i].name,s)==0)
            {
                return ST[i].type[0];
            }
        }

}

void printST()
{
     printf("%15s | %13s | %10s | %9s | %10s | %10s | %13s |\n","NAME",
"CLASS", "TYPE","SCOPE", "LINE NO", "VALUE", "NO OF PARAMS");
    for(int i=0;i<100;i++) {
        printf("-");
    }
    printf("\n");
    for(int i = 0 ; i < 1001 ; i++)
    {
            if(ST[i].length == 0 || strcmp(ST[i].class, "Keyword") == 0 ||
strcmp(ST[i].name, "printf") == 0)
```

```
        {
            continue;
        }

        if(strcmp(ST[i].class, "Identifier") == 0)
                printf("%15s | %13s | %10s | %9d | %10d | %10s | %13c
|\n",ST[i].name,  ST[i].class,  ST[i].type,  ST[i].scope,  ST[i].lineno,
ST[i].value, '-');
        else if(strcmp(ST[i].class, "Array") == 0)
        {
                printf("%15s | %13s | %10s | %9d | %10d | %10c | %13c
|\n",ST[i].name, ST[i].class, ST[i].type, ST[i].scope, ST[i].lineno, '-',
'-');
        }
        else if(strcmp(ST[i].class, "Function") == 0)
        {
                printf("%15s | %13s | %10s | %9d | %10d | %10c | %13d
|\n",ST[i].name, ST[i].class, ST[i].type, ST[i].scope, ST[i].lineno, '-',
ST[i].params_count);
        }



    }

printf("-----------------------------------------------------------------------
---------------------------------\n");
}


void printCT()
{
   printf("%10s | %25s |\n","NAME", "TYPE");
   printf("----------------------------------------");
   printf("\n");
   for(int i = 0 ; i < 1001 ; i++)
   {
        if(CT[i].length == 0)
            continue;

        printf("%10s | %25s |\n",CT[i].name, CT[i].type);
   }
   printf("----------------------------------------\n\n");
}
```

2.4 Utility Functions

utils.h

```
char* itoa(int num, char* str, int base);

struct stack
{
   char value[100];
   int labelvalue;
}s[100],label[100];

extern char curid[20];
extern char curtype[20];
extern char curval[20];
extern int currnest;
void deletedata (int );
char currfunctype[100];
char currfunc[100];
char currfunccall[100];



int top = 0,count=0,ltop=0,lno=0;
char temp[3] = "t";
void label1()
{
   strcpy(temp,"L");
   char buffer[100];
   itoa(lno,buffer,10);
   strcat(temp,buffer);
   printf("IF not %s GoTo %s\n",s[top].value,temp);
   label[++ltop].labelvalue = lno++;
}

void label2()
{
   strcpy(temp,"L");
   char buffer[100];
   itoa(lno,buffer,10);
   strcat(temp,buffer);
   printf("GoTo %s\n",temp);
   strcpy(temp,"L");
```

```
    itoa(label[ltop].labelvalue,buffer,10);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    ltop--;
    label[++ltop].labelvalue=lno++;
}

void label3()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(label[ltop].labelvalue,buffer,10);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    ltop--;

}

void label4()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(lno,buffer,10);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    label[++ltop].labelvalue = lno++;
}


void label5()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(label[ltop-1].labelvalue,buffer,10);
    strcat(temp,buffer);
    printf("GoTo %s:\n",temp);
    strcpy(temp,"L");
    itoa(label[ltop].labelvalue,buffer,10);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    ltop = ltop - 2;
```

```
}

void push(char *x)
{
   strcpy(s[++top].value,x);
}

void swap(char *x, char *y)
{
   char temp = *x;
   *x = *y;
   *y = temp;
}

void reverse(char str[], int length)
{
   int start = 0;
   int end = length -1;
   while (start < end)
   {
       swap((str+start), (str+end));
       start++;
       end--;
   }
}
 char* itoa(int num, char* str, int base)
{
   int i = 0;
   int isNegative = 0;

   if (num == 0)
   {
       str[i++] = '0';
       str[i] = '\0';
       return str;
   }
    if (num < 0 && base == 10)
   {
       isNegative = 1;
       num = -num;
   }

   while (num != 0)
   {
       int rem = num % base;
       str[i++] = (rem > 9)? (rem-10) + 'a' : rem + '0';
       num = num/base;
   }
    if (isNegative)
       str[i++] = '-';
    str[i] = '\0';

   reverse(str, i);
    return str;
}

void codegen()
{
   strcpy(temp,"t");
   char buffer[100];
   itoa(count,buffer,10);
   strcat(temp,buffer);
                              printf("%s          =          %s          %s
%s\n",temp,s[top-2].value,s[top-1].value,s[top].value);
   top = top - 2;
   strcpy(s[top].value,temp);
   count++;
}

void codegencon()
{
   strcpy(temp,"t");
   char buffer[100];
   itoa(count,buffer,10);
   strcat(temp,buffer);
   printf("%s = %s\n",temp,curval);
   push(temp);
   count++;

}

int isunary(char *s)
{
   if(strcmp(s, "--")==0 || strcmp(s, "++")==0)
   {
       return 1;
   }
   return 0;
}

void genunary()
{
   char temp1[100], temp2[100], temp3[100];
   strcpy(temp1, s[top].value);
   strcpy(temp2, s[top-1].value);

   if(isunary(temp1))
   {
       strcpy(temp3, temp1);
       strcpy(temp1, temp2);
       strcpy(temp2, temp3);
   }
   strcpy(temp, "t");
   char buffer[100];
   itoa(count, buffer, 10);
   strcat(temp, buffer);
   count++;

   if(strcmp(temp2,"--")==0)
   {
       printf("%s = %s - 1\n", temp, temp1);
       printf("%s = %s\n", temp1, temp);
   }

   if(strcmp(temp2,"++")==0)
   {
       printf("%s = %s + 1\n", temp, temp1);
       printf("%s = %s\n", temp1, temp);
   }

   top = top -2;
}

void codeassign()
{
   printf("%s = %s\n",s[top-2].value,s[top].value);
   top = top - 2;
}
```

```
void funcgen()
{
   printf("func begin %s\n",currfunc);
}

void funcgenend()
{
   printf("func end\n\n");
}

void arggen(int i)
{
   if(i==1)
   {
   printf("refparam %s\n", curid);
   }
   else
   {
   printf("refparam %s\n", curval);
   }
}
```

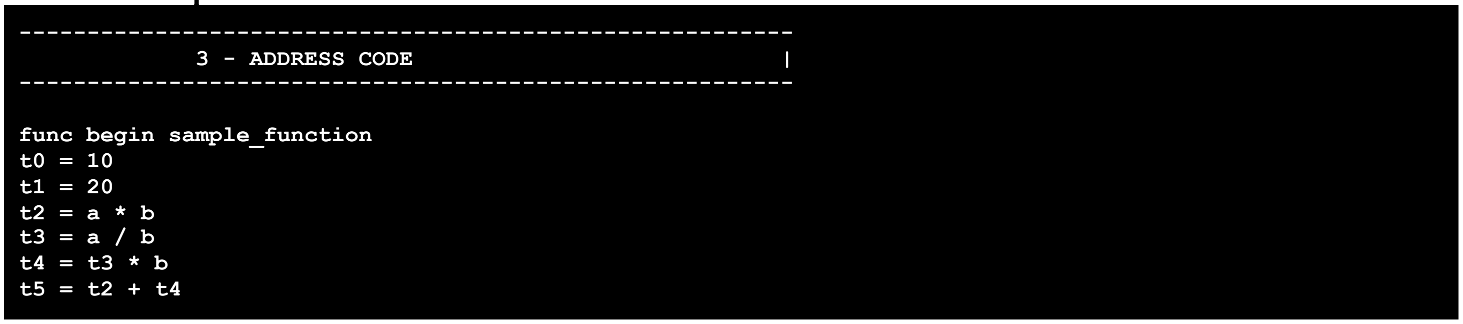
3. Test Cases

3.1 Without Errors:

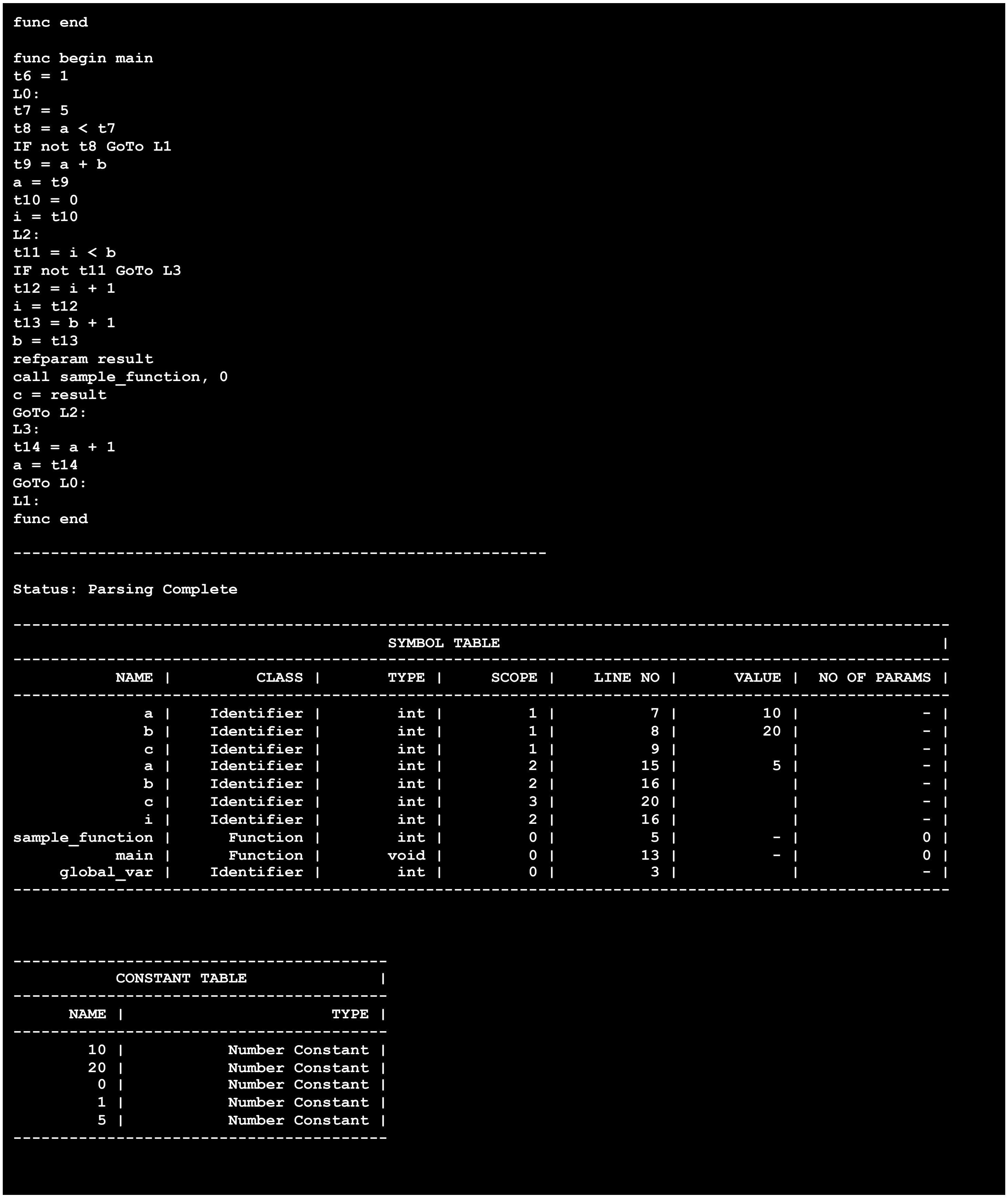
Test 1 :

```
#include <stdio.h>

int global_var;

int sample_function()
{
   int a = 10;
   int b = 20;
   int c = (a * b) + (a / b) * b;
   return c;
}

void main()
{
   int a = 1;
   int b,i;

   while(a < 5)
   {
       int c;
       a = a+b;
       for(i=0;i<b;i++)
       {
           b++;
           c = sample_function();
       }
       a++;
   }

}
```

Output :



```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin sample_function
t0 = 10
t1 = 20
t2 = a * b
t3 = a / b
t4 = t3 * b
t5 = t2 + t4
```

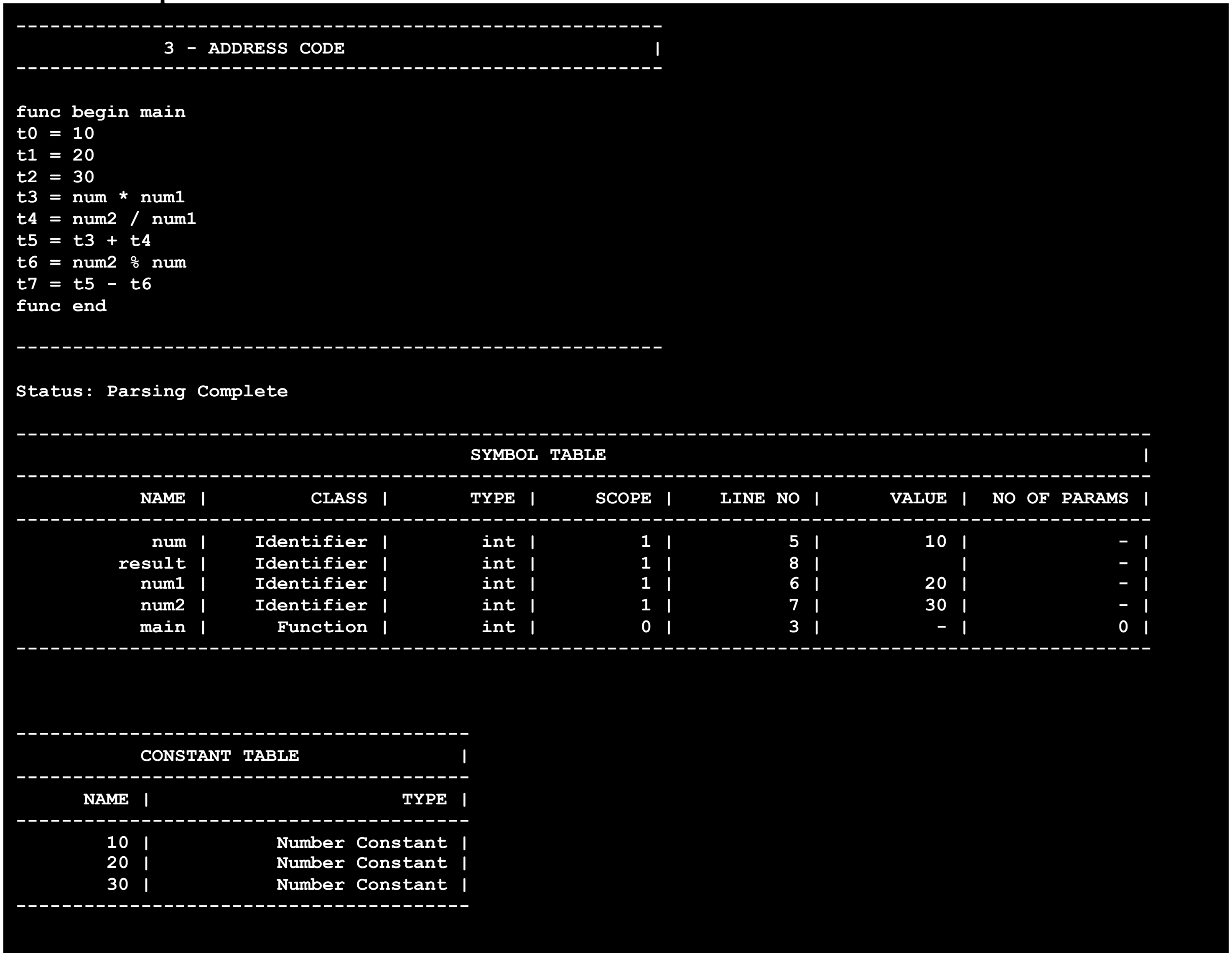


```
func end

func begin main
t6 = 1
L0:
t7 = 5
t8 = a < t7
IF not t8 GoTo L1
t9 = a + b
a = t9
t10 = 0
i = t10
L2:
t11 = i < b
IF not t11 GoTo L3
t12 = i + 1
i = t12
t13 = b + 1
b = t13
refparam result
call sample_function, 0
c = result
GoTo L2:
L3:
t14 = a + 1
a = t14
GoTo L0:
L1:
func end

----------------------------------------------------------

Status: Parsing Complete

-----------------------------------------------------------------------------------------------------
                                        SYMBOL TABLE                                                |
-----------------------------------------------------------------------------------------------------
           NAME |         CLASS |       TYPE |      SCOPE |    LINE NO |       VALUE |  NO OF PARAMS |
-----------------------------------------------------------------------------------------------------
              a |    Identifier |        int |          1 |          7 |          10 |             - |
              b |    Identifier |        int |          1 |          8 |          20 |             - |
              c |    Identifier |        int |          1 |          9 |             |             - |
              a |    Identifier |        int |          2 |         15 |           5 |             - |
              b |    Identifier |        int |          2 |         16 |             |             - |
              c |    Identifier |        int |          3 |         20 |             |             - |
              i |    Identifier |        int |          2 |         16 |             |             - |
sample_function |      Function |        int |          0 |          5 |           - |             0 |
           main |      Function |       void |          0 |         13 |           - |             0 |
     global_var |    Identifier |        int |          0 |          3 |             |             - |
-----------------------------------------------------------------------------------------------------



----------------------------------------
           CONSTANT TABLE              |
----------------------------------------
      NAME |                     TYPE |
----------------------------------------
        10 |          Number Constant |
        20 |          Number Constant |
         0 |          Number Constant |
         1 |          Number Constant |
         5 |          Number Constant |
----------------------------------------
```

Test 2:

```
#include<stdio.h>

int main()
{
   int num = 10;
   int num1 = 20;
   int num2 = 30;
   int result = (num * num1) + (num2 / num1) - (num2 % num);
}
```

Output:



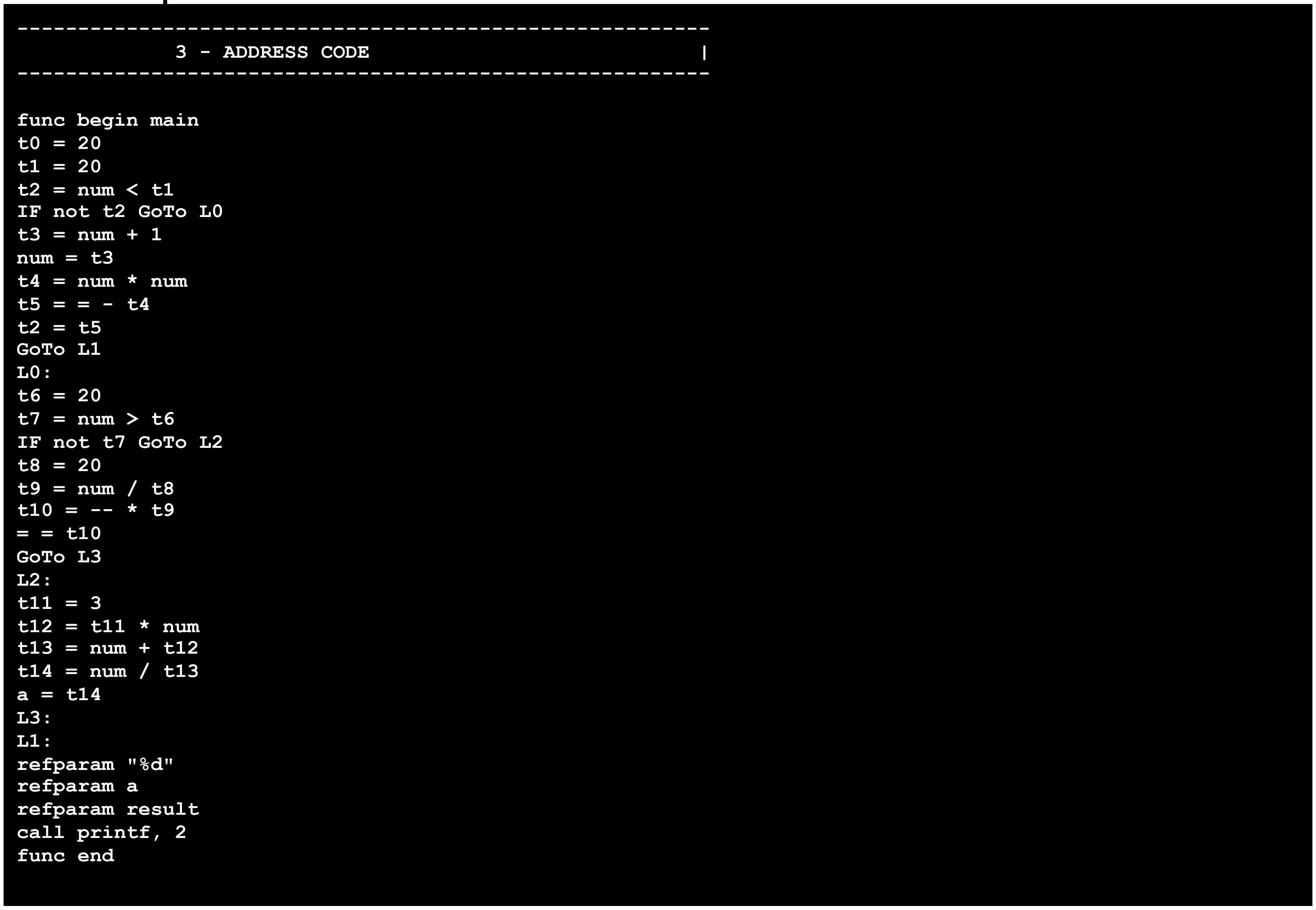
```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin main
t0 = 10
t1 = 20
t2 = 30
t3 = num * num1
t4 = num2 / num1
t5 = t3 + t4
t6 = num2 % num
t7 = t5 - t6
func end

----------------------------------------------------------

Status: Parsing Complete

----------------------------------------------------------------------------------------------------
                                        SYMBOL TABLE                                               |
----------------------------------------------------------------------------------------------------
           NAME |           CLASS |         TYPE |      SCOPE |     LINE NO |       VALUE |  NO OF PARAMS |
----------------------------------------------------------------------------------------------------
            num |     Identifier |          int |          1 |           5 |          10 |             - |
         result |     Identifier |          int |          1 |           8 |             |             - |
           num1 |     Identifier |          int |          1 |           6 |          20 |             - |
           num2 |     Identifier |          int |          1 |           7 |          30 |             - |
           main |       Function |          int |          0 |           3 |           - |             0 |
----------------------------------------------------------------------------------------------------



----------------------------------------
           CONSTANT TABLE              |
----------------------------------------
      NAME |                     TYPE |
----------------------------------------
        10 |          Number Constant |
        20 |          Number Constant |
        30 |          Number Constant |
----------------------------------------
```

Test 3:

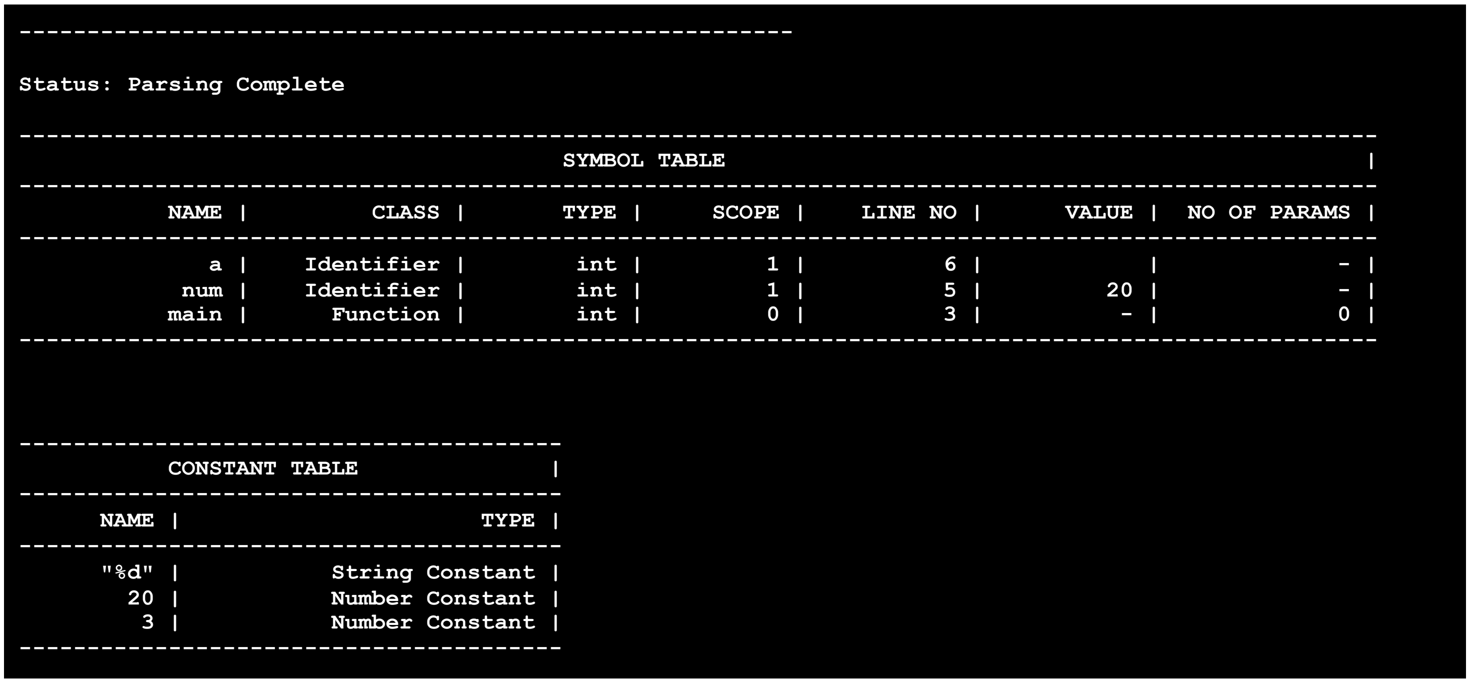
```
#include <stdio.h>
```

```c
int main()
{
   int num = 20;
   int a;
   if(num < 20)
   {
       a = (num++) - (num * num);
   }
   else if(num > 20)
   {
       a = (num--) * (num / 20);
   }
   else
   {
       a = num / (num + (3*num));
   }

   printf("%d", a);

}
```

Output :



```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin main
t0 = 20
t1 = 20
t2 = num < t1
IF not t2 GoTo L0
t3 = num + 1
num = t3
t4 = num * num
t5 = = - t4
t2 = t5
GoTo L1
L0:
t6 = 20
t7 = num > t6
IF not t7 GoTo L2
t8 = 20
t9 = num / t8
t10 = -- * t9
= = t10
GoTo L3
L2:
t11 = 3
t12 = t11 * num
t13 = num + t12
t14 = num / t13
a = t14
L3:
L1:
refparam "%d"
refparam a
refparam result
call printf, 2
func end
```



```
----------------------------------------------------------

Status: Parsing Complete

----------------------------------------------------------------------------------------------------
                                        SYMBOL TABLE                                               |
----------------------------------------------------------------------------------------------------
          NAME |           CLASS |          TYPE |       SCOPE |      LINE NO |         VALUE |  NO OF PARAMS |
----------------------------------------------------------------------------------------------------
             a |      Identifier |           int |           1 |            6 |               |             - |
           num |      Identifier |           int |           1 |            5 |            20 |             - |
          main |        Function |           int |           0 |            3 |             - |             0 |
----------------------------------------------------------------------------------------------------



----------------------------------------
          CONSTANT TABLE              |
----------------------------------------
     NAME |                      TYPE |
----------------------------------------
     "%d" |           String Constant |
       20 |           Number Constant |
        3 |           Number Constant |
----------------------------------------
```

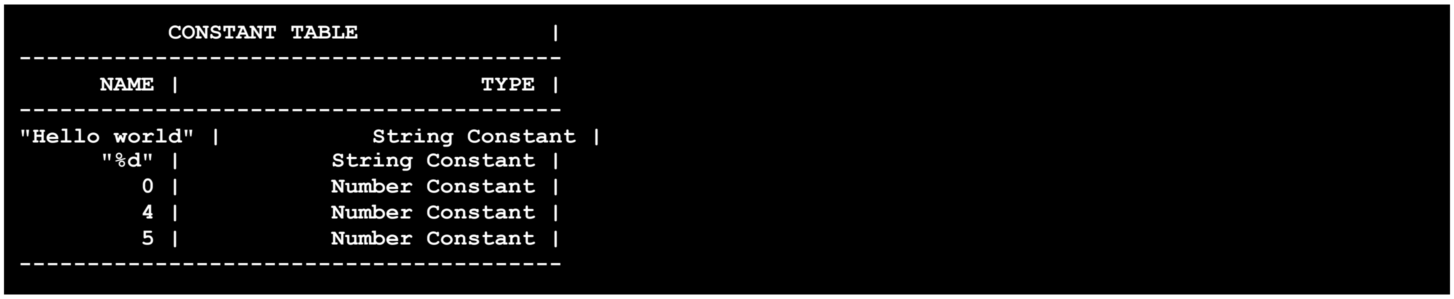
Test 4:

```
#include<stdio.h>

int main()
{
   int a = 5;
   while(a>0)
   {
       printf("Hello world");
       a--;
   }

   a=4;
   while(a>0)
   {
       printf("%d",a);
       a--;
       int b;
       b= 4;
       while(b>0)
       {
           printf("%d", b);
```

```
            b--;
        }
    }
}
```

Output :

```
---------------------------------------------------------
             3 - ADDRESS CODE                           |
---------------------------------------------------------

func begin main
t0 = 5
L0:
t1 = 0
t2 = a > t1
IF not t2 GoTo L1
refparam "Hello world"
refparam result
call printf, 1
GoTo L0:
L1:
t3 = 4
a = t3
L2:
t4 = 0
t5 = a > t4
IF not t5 GoTo L3
refparam "%d"
refparam a
refparam result
call printf, 2
t6 = 4
b = t6
L4:
t7 = 0
t8 = b > t7
IF not t8 GoTo L5
refparam "%d"
refparam b
refparam result
call printf, 2
GoTo L4:
L5:
GoTo L2:
L3:
func end

---------------------------------------------------------

Status: Parsing Complete

----------------------------------------------------------------------------------------------------
                                        SYMBOL TABLE                                               |
----------------------------------------------------------------------------------------------------
           NAME |          CLASS |          TYPE |      SCOPE |     LINE NO |       VALUE |  NO OF PARAMS |
----------------------------------------------------------------------------------------------------
              a |     Identifier |           int |          1 |           5 |           0 |             - |
              b |     Identifier |           int |          3 |          17 |           0 |             - |
           main |       Function |           int |          0 |           3 |           - |             0 |
----------------------------------------------------------------------------------------------------



----------------------------------------
```

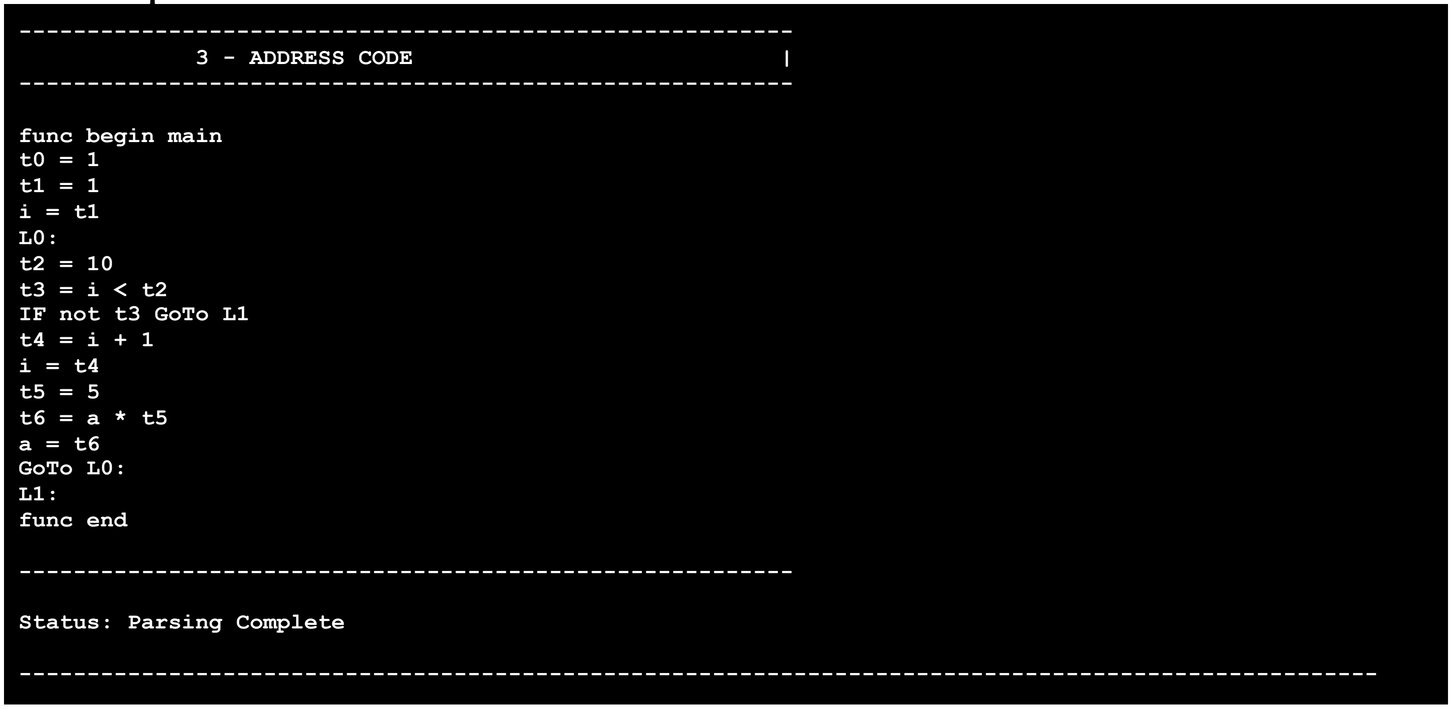


```
            CONSTANT TABLE                |
----------------------------------------
      NAME |                        TYPE |
----------------------------------------
"Hello world" |             String Constant |
      "%d" |             String Constant |
         0 |             Number Constant |
         4 |             Number Constant |
         5 |             Number Constant |
----------------------------------------
```

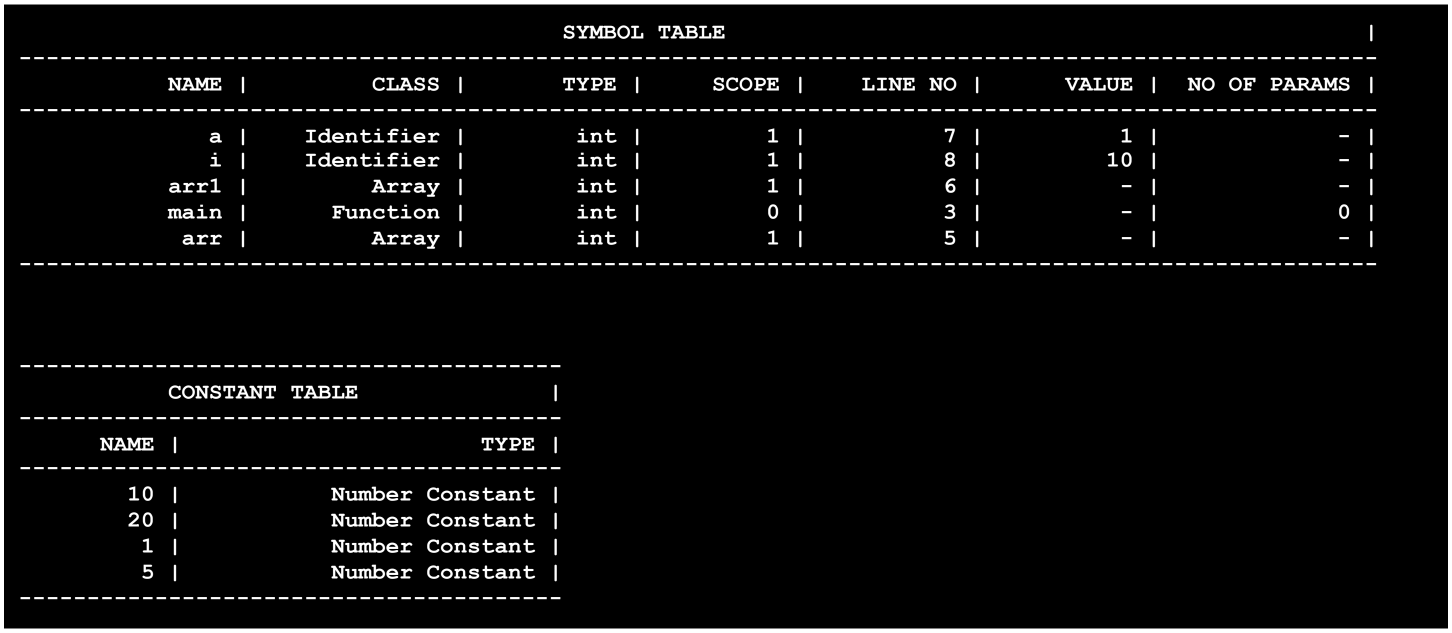
Test 5:

```
#include<stdio.h>

int main()
{
   int arr[10];
   int arr1[20];
   int a = 1;
   int i;
   for(i=1; i<10; i++)
   {
       a = a * 5;
   }
}
```

Output:



```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin main
t0 = 1
t1 = 1
i = t1
L0:
t2 = 10
t3 = i < t2
IF not t3 GoTo L1
t4 = i + 1
i = t4
t5 = 5
t6 = a * t5
a = t6
GoTo L0:
L1:
func end

----------------------------------------------------------

Status: Parsing Complete

----------------------------------------------------------------------------------------------------
```



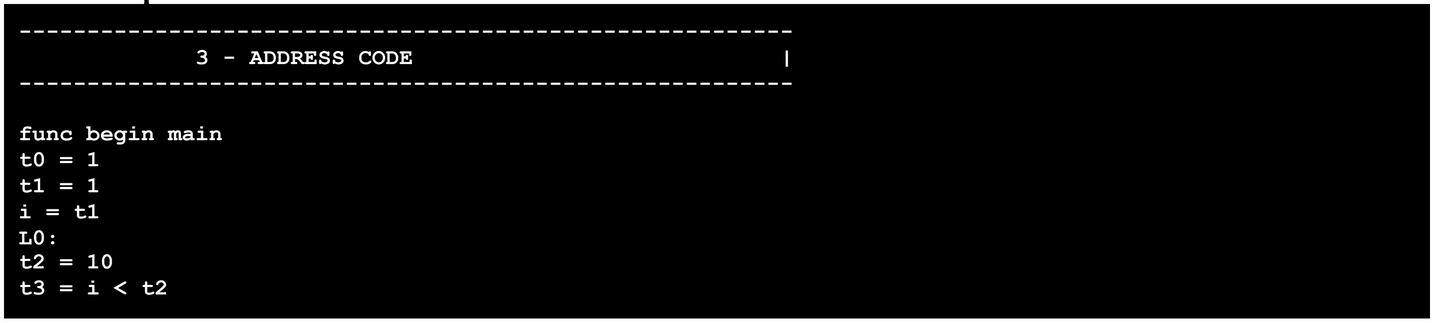
```
                                        SYMBOL TABLE                                        |
-----------------------------------------------------------------------------------------------------
          NAME |           CLASS |         TYPE |       SCOPE |     LINE NO |       VALUE |  NO OF PARAMS |
-----------------------------------------------------------------------------------------------------
             a |      Identifier |          int |           1 |           7 |           1 |             - |
             i |      Identifier |          int |           1 |           8 |          10 |             - |
          arr1 |           Array |          int |           1 |           6 |           - |             - |
          main |        Function |          int |           0 |           3 |           - |             0 |
           arr |           Array |          int |           1 |           5 |           - |             - |
-----------------------------------------------------------------------------------------------------



----------------------------------------
          CONSTANT TABLE               |
----------------------------------------
     NAME |                       TYPE |
----------------------------------------
       10 |            Number Constant |
       20 |            Number Constant |
        1 |            Number Constant |
        5 |            Number Constant |
----------------------------------------
```

3.2 With Errors

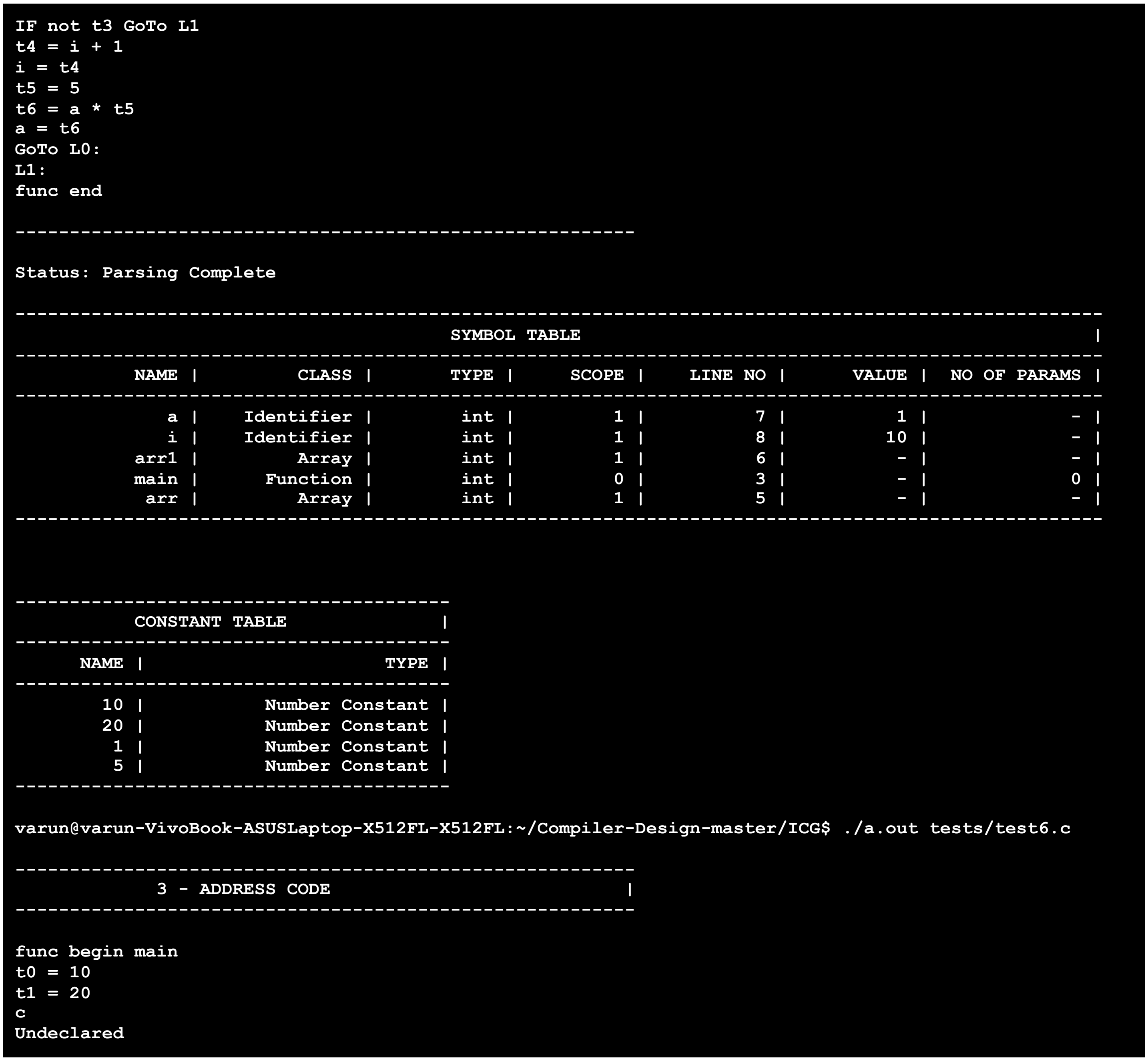
Test 6:

```
// Program having undeclared variable error

#include <stdio.h>

int main()
{
   int a = 10;
   int b = 20;
   c = a + b;
   printf("%d", c);

   return 0;
}
```

Output:



```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin main
t0 = 1
t1 = 1
i = t1
L0:
t2 = 10
t3 = i < t2
```



```
IF not t3 GoTo L1
t4 = i + 1
i = t4
t5 = 5
t6 = a * t5
a = t6
GoTo L0:
L1:
func end

----------------------------------------------------------

Status: Parsing Complete

----------------------------------------------------------------------------------------------------------
                                        SYMBOL TABLE                                                   |
----------------------------------------------------------------------------------------------------------
           NAME |           CLASS |          TYPE |         SCOPE |      LINE NO |          VALUE |  NO OF PARAMS |
----------------------------------------------------------------------------------------------------------
              a |      Identifier |           int |             1 |            7 |              1 |             - |
              i |      Identifier |           int |             1 |            8 |             10 |             - |
           arr1 |           Array |           int |             1 |            6 |              - |             - |
           main |        Function |           int |             0 |            3 |              - |             0 |
            arr |           Array |           int |             1 |            5 |              - |             - |
----------------------------------------------------------------------------------------------------------



----------------------------------------
           CONSTANT TABLE              |
----------------------------------------
      NAME |                      TYPE |
----------------------------------------
        10 |           Number Constant |
        20 |           Number Constant |
         1 |           Number Constant |
         5 |           Number Constant |
----------------------------------------

varun@varun-VivoBook-ASUSLaptop-X512FL-X512FL:~/Compiler-Design-master/ICG$ ./a.out tests/test6.c

----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin main
t0 = 10
t1 = 20
c
Undeclared
```

Test 7:

```
// Program having type mismatch error


#include <stdio.h>


int main()
```

```
{
   int a = 10;
   float b;
   float c;
   c = a + b;

   return 0;
}
```

Output:

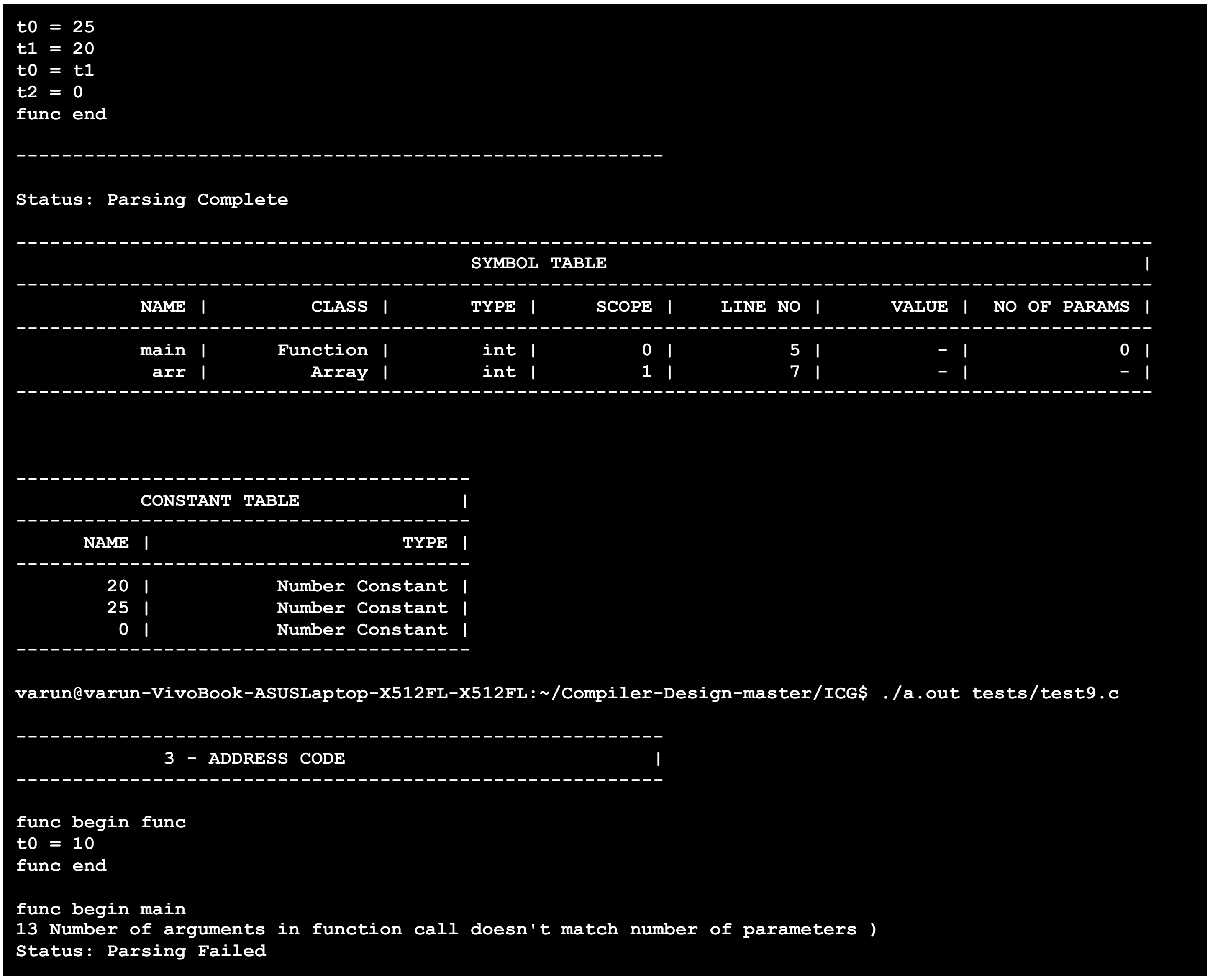
```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin main
t0 = 10
t1 = a + b
Type mismatch
```

Test 9:

```
// Function call parameter list not matching function signature

#include <stdio.h>

void func(int a)
{
   int b = 10;
   return;
}

int main()
{
   func();
   return 0;
}
```

Output:

```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin main
```



```
t0 = 25
t1 = 20
t0 = t1
t2 = 0
func end

-----------------------------------------------------------

Status: Parsing Complete

---------------------------------------------------------------------------------------------------------
                                        SYMBOL TABLE                                                    |
---------------------------------------------------------------------------------------------------------
           NAME |           CLASS |          TYPE |       SCOPE |      LINE NO |        VALUE |  NO OF PARAMS |
---------------------------------------------------------------------------------------------------------
           main |        Function |           int |           0 |            5 |            - |             0 |
            arr |           Array |           int |           1 |            7 |            - |             - |
---------------------------------------------------------------------------------------------------------



-----------------------------------------
           CONSTANT TABLE               |
-----------------------------------------
      NAME |                      TYPE |
-----------------------------------------
        20 |           Number Constant |
        25 |           Number Constant |
         0 |           Number Constant |
-----------------------------------------

varun@varun-VivoBook-ASUSLaptop-X512FL-X512FL:~/Compiler-Design-master/ICG$ ./a.out tests/test9.c

-----------------------------------------------------------
             3 - ADDRESS CODE                             |
-----------------------------------------------------------

func begin func
t0 = 10
func end

func begin main
13 Number of arguments in function call doesn't match number of parameters )
Status: Parsing Failed
```

Test 10:

```
// Redeclaration of variable in the same scope


#include <stdio.h>


int main()

{

   int a = 10;
   printf("%d", a);
   int a = 40;
   printf("%d", a);

   return 0;
}
```

Output:

```
-----------------------------------------------------------
             3 - ADDRESS CODE                             |
-----------------------------------------------------------

func begin main
t0 = 10
refparam "%d"
refparam a
refparam result
call printf, 2
Duplicate
```

Test 11:

```
// Variable out of scope

#include <stdio.h>

int main()
{
   {
       int a = 10;
       int b = 20;
       printf("a is %d\n", a);
       printf("b is %d\n", b);
   }
   a = 40;

   return 0;
}
```

Output:

```
-----------------------------------------------------------
             3 - ADDRESS CODE                             |
-----------------------------------------------------------
```

```
func begin main
t0 = 10
t1 = 20
refparam "a is %d\n"
refparam a
refparam result
call printf, 2
refparam "b is %d\n"
refparam b
refparam result
call printf, 2
a
Undeclared
```

Test 12:

```c
// Use of non-array variable with subscript

#include <stdio.h>

int main()
{
   int a;
   a[0] = 40;

   return 0;
}
```

Output:

```
----------------------------------------------------------
            3 - ADDRESS CODE                             |
----------------------------------------------------------

func begin main
Error use of array
```

Test 13:

```c
// Use of array variable without subscript

#include <stdio.h>

int main()
{
   int a[10];
   a = 40;

   return 0;
}
```

Output:

```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin main
a
Array ID has no subscript
```

Test 14:

```
// Use of array of size less than 1

#include <stdio.h>

int main()
{
   int a[0];

   return 0;
}
```

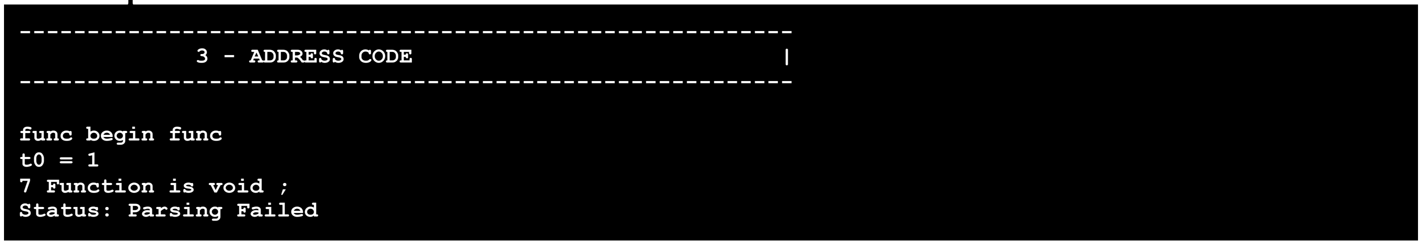
Output:

```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin main
Wrong array size
```

Test 15:

```
// A void function trying to return a value

#include <stdio.h>

void func()
{
   return 1;
}


int main()
{
   int a = 10;


   return 0;
}
```

Output:



```
----------------------------------------------------------
             3 - ADDRESS CODE                            |
----------------------------------------------------------

func begin func
t0 = 1
7 Function is void ;
Status: Parsing Failed
```

4. Implementation Details

The parser implements C grammar using many production rules. The parser takes tokens from the lexer output, one at a time, and applies the corresponding production rules to generate the symbol table with the variable and function types. In addition to the previous work semantic rules are defined in the lexer. If the parsing is not successful, the parser outputs the line number with the corresponding error.

The Symbol Table are updated with:

- Data Type of each identifier.
- The value associated with in the case of non-functional identifiers.
- The line number of the declaration of each identifier.
- The scope associated with the identifier.
- Number of parameters in the case of functional identifiers.

The Constant Table is updated with the name and type of constant encountered.

Additional Semantic rules are added to the grammar to generate the three Address Code.
These Semantic rules make use of the functions described below.

Function Name	**Description**
`funcgen`	It indicates the beginning of a function.
`funcgenend`	It indicates the ending of a function.
`arggen`	It displays all the reference parameters that are used in a function call.
`callgen`	It calls the function i.e. displays the appropriate function call according to the three address code.
`itoa`	It worked as a utility function since we had to name temporary variables and labels. It was used to convert int to string and used several functions like reverse, swap to do it.
`codeassign`	This function is specifically designed for assignment operators. It assigns the final temp variable value (after all the evaluation) to the desired variable.
`label1`	It is used while evaluating conditions of loops or if statements. If the condition is not satisfied then it states where to jump to i.e. on which label the control should go.
`label2`	It is used when the statement block pertaining to if statement is over. It tells where the control flow should go once that block is over i.e. it jumps the else statement block.
`label3`	It is used after the whole if else construct is over. It gives label that tells where to jump after the if block is executed.
`label4`	It is used to give labels to starting loops.
`label5`	It is used after the statement block of the loop. It indicates the label to jump to and also generates the label where the control should go once the loop is terminated.
`codegen`	This function is called whenever a reduction of an expression takes place. It creates the temporary variable and displays the desired 3 address code i.e x = y op z.
`codegencon`	This function is especially written for reductions of expression involving constants since its 3 address code is x op z.
`isunary`	This function checks if the operator is an unary operator like ‘++’. If so it returns true else false.

`genunary`	This function is specifically designed to generate 3 address code for unary operations. It makes use of the isuanary() function mentioned above.

5. Results and Future Work

In this final phase of the compiler design project, we have built the frontend of a mini C compiler that successfully produces the three address representation as the intermediate code and prints the symbol table and constant table. The parser also generates error messages in case of any syntactical errors or semantic errors in the test program.

However in our implementation, we only check for the most common of semantic rules that are to be followed in C source code and the generated intermediate code is simple. There is a lot of literature available for optimizing compilers and optimizations can be implemented at all the phases, from the parser all the way upto the ICG and the backend as well. We could later try adopting some of these techniques in our implementation.

6. References

1. A. Aho, M. Lam, R. Sethi and J. Ullman, “Compilers Principles, Tools and Techniques”
2. https://silcnitc.github.io/yacc.html
3. https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf
4. https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dgt/index.html
5. https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm
6. https://www.geeksforgeeks.org/three-address-code-compiler/