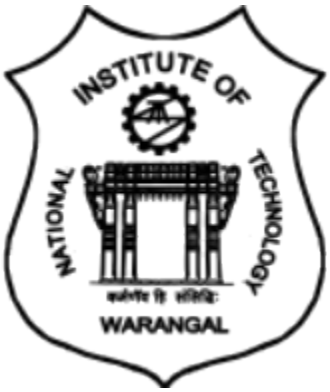
THREE-ADDRESS INTERMEDIATE CODE GENERATION

USING LEX AND YACC



BY

RAJENDRA KUJUR

ROLL NO: 157150

SECTION: A

3rd Year, B.Tech

Computer Science & Engineering

NIT, Warangal

TABLE OF CONTENTS

PROBLEM DESCRIPTION

This project aims at generating three-address intermediate code for a given Sub-C code, along with syntax and semantic analysis, using *lex* as the lexical analyzing tool and *yacc* as the parsing tool.

The Sub-C language is a subset of C language and contains following language constructs, which are to be tested for while parsing the given Sub-C code:

DATA TYPES

- Integer (int)
- Unsigned Integer (uint)
- Boolean (bool)

KEYWORDS

- ➢ ➢ ➢ CONDITIONAL CONSTRUCTS
  - if
  - else
  - switch
  - case
  - break
  - default
- ➢ ➢ ➢ ITERATIVE CONSTRUCTS
  - while
- ➢ ➢ ➢ BOOLEAN LITERALS
  - true
  - false

OPERATORS

- ASSIGNMENT OPERATORS: =, +=, -=, *=, /=
- BINARY OPERATORS: +, -, *, /, %
- EXPONENTIATION: @
- BITWISE OPERATORS: &, |, ^, ~
- LOGICAL OPERATORS: &&, ||, !
- RELATIONAL OPERATORS: ==, !=, <, <=, >, >=
- BRACES: (), {}

IDENTIFIERS

- [a-zA-Z]+

DECOMPOSITION

The entire project can be decomposed into following tasks:

- **LEXICAL ANALYSIS:** Using lex tool, the given Sub-C code is tokenized into stream of tokens and passed on to the parser for the next phase.
- **PARSING:** The received stream of tokens is parsed for syntax analysis by checking with the defined grammar rules. This is done using yacc tool.
- **INTERMEDIATE CODE GENERATION:** The action part of each grammar production is used for generating the corresponding three-address intermediate code, which is appended with the entire code in a bottom-up fashion.

SOLUTION APPROACH & ALGORITHM PROPOSED

The task of three-address intermediate code generation for a given Sub-C code snippet is accomplished using lex and yacc tools. The code is first broken down into set of tokens using lex. This set of tokens is then passed on to yacc which checks the syntax of the code. Also, yacc file contains the steps for generating the three-address intermediate code for each Sub-C statement, which is concatenated in a bottom-up manner as parsing of each statement proceeds. This intermediate code is stored in a global buffer which is printed as the final output in the end.

There exists a structure associated with each variable of the grammar. This structure consists of fields such as code, begin label, after label, place etc. which are needed for the intermediate code generation.

There also exists a global symbol table, which is nothing but a mapping for identifiers and their respective data types, for type checking and semantic checking. This allows the parser to detect cases when a variable is being used without being declared previously and also when a variable of incompatible type is being used in a statement.

DESCRIPTION OF FUNCTIONAL UNITS

Following functional units have been used in the lex file:

- void concat (int n, ...)

This function takes a variable number of arguments. The count of inputs is given by the first argument (n). This is followed by a target character string n-1 character strings which are concatenated into the target string.

- void init_int(char*t1, char*t2, char*code, char*c1, char*c2, char*place, char*type, bool term1, bool term2)

While generating the intermediate code for expressions (involving integers), we need to do type compatibility checking and some other initializations. This function performs this task, and thus is called in the beginning of action part of each production for expressions.

- void init_bool(char*t1, char*t2, char*code, char*c1, char*c2, char*place, char*type, bool term1, bool term2, char t)

This function is same as init_int. The only difference is that it is used while generating intermediate code for expressions involving booleans.

- int get_free_map()

The switch statements require the use of a map which needs to be global. This function returns index to a free map from a pool of maps.

- char* getNewId(char* id)

Each time a temporary identifier is required in the intermediate code, this function is called.

- char* getNewLabel(char* id)

This function returns a unique label each time it’s called.

- void itoa(char* ret,int n)

This function converts integer n into a character string and stores it in the first argument of the function (ret).

IMPLEMENTATION DETAILS OF FUNCTIONAL UNITS

```
void concat(int size,...)
{
   va_list varlist;
   va_start(varlist,size);
   char* target = va_arg(varlist,char*);
   for(int i=1;i<size;i++)
   {

   strcat(target,va_arg(varlist,char*));
   }
}
```

This function takes variable number of inputs, count of which is given by the first argument size. The target character string concatenates all the remaining (n-1) character strings in a for loop running n-1 times.

```
void init_int(char*t1, char*t2, char*code, char*c1,
char*c2, char*place, char*type, bool term1, bool term2)
{
     if(strcmp(t1,"bool")==0 || strcmp(t2,"bool")==0)
     {
          cout<<"Incompatible types (init_int)!"<<endl;
```

This function first checks the type of both the operands. If either of them is found to be a bool, it's reported as an error. If the type is compatible, then a new entry is created in the symbol table for an int identifier. Also, if there's a code is associated with any of the operand expressions, it gets appended with the resulting operand.

```
void init_bool(char*t1, char*t2,char*code, char*c1,
char*c2, char*place, char*type, bool term1, bool term2,
char t)
{
     if(t=='b')
     {
          if(strcmp(t1,"bool")!=0 || strcmp(t2,"bool")!=0)
          {
```

This function is implemented in the same way as init_int, the only difference being that it’s done for boolean operands.

```
int get_free_map()
{
     int index = *free_map_indices.begin();
```

This function finds the first free map, removes it from the set of unused maps, and returns it's index.

```
char* getNewId(char* id)
{
      strcpy(id,"_t");
      itoa(id+2,cur_id);
      cur_id++;
      return id;
}
```

The intermediate code requires temporary identifiers. This function creates a character string, starting with "_t" and appends it with the current identifier number stored in a variable cur_id. The variable cur_id is incremented at the end, so that next time this function is called, a new identifier is generated.

```
char* getNewLabel(char* id)
{
      strcpy(id,"_L");
      itoa(id+2,cur_lab);
      cur_lab++;
      return id;
}
```

This function creates a character string starting with “_L” and appends it with the current label number stored in a variable called cur_lab. This string is then returned by the function to be used as a label in the intermediate code. The variable cur_lab is incremented at the end of this function to generate a new label next time.

```
void itoa(char* ret,int n)
{
     int i=0;
     while(n)
     {
          ret[i++] = (n%10+'0');
          n/=10;
     }
     for(int j=0;j<i/2;j++)
     {
```

This function is used for converting an integer n, into a character string (ret) by repeatedly appending the least significant digit of n to ret. The string ret now contains the number in reverse order, hence it needs to be reversed which is done in the next loop.

CODE

LEX SPECIFICATION FILE

```
%{
#include "major.tab.h"
#include <bits/stdc++.h>
using namespace std;
%}

%%
"if"                { return IF; }
"else"                   { return ELSE; }
"while"                  { return WHILE; }
"switch"            { return SWITCH; }
"break"                  { return BREAK; }
"default"           { return DEFAULT; }
"case"                   { return CASE; }
"true"                   { return TR; }
"false"                  { return FL; }
"int"                    { return INT; }
"uint"                   { return UINT; }
"bool"                   { return BOOL; }
[0-9]+                   { yylval.str =
(char*)malloc(strlen(yytext)+1); strcpy(yylval.str,yytext);
return NUM; }
```

```
[a-zA-Z]+           { yylval.str =
(char*)malloc(strlen(yytext)+1); strcpy(yylval.str,yytext);
return ID; }

"+="                { return ADDASGN; }

"-="                { return SUBASGN; }

"*="                { return MULASGN; }

"/="                { return DIVASGN; }

"+"                     { return ADD; }

"-"                     { return SUB; }

"*"                     { return MUL; }

"/"                     { return DIV; }

"%"                     { return MOD; }

"@"                     { return POW; }

"|"                     { return OR; }

"&"                     { return AND; }

"^"                     { return XOR; }

"~"                     { return NOT; }

"||"                { return LOR; }

"&&"                { return LAND; }

"=="                { return EQ; }

"!="                { return NEQ; }

"<"                     { return LT; }

"<="                { return LTEQ; }

">"                     { return GT; }

">="                { return GTEQ; }

"="                     { return ASGN; }
```

```
[ \t\n]+            {}
"("                      { return LP; }
")"                      { return RP; }
"{"                      { return LC; }
"}"                      { return RC; }
";"                      { return TERM; }
":"                      { return CLN; }
.                        { return yytext[0]; }
%%
```

YACC SPECIFICATION FILE

```
%{

#include <bits/stdc++.h>

using namespace std;

int yylex();

void yyerror(const char*);

char* getNewId(char*);                          //generates a new
identifier for temporary variables

char* getNewLabel(char*);                       //generates a new
label

void itoa(char*,int);

void concat(int,...);                           //concatenates n
number of character strings where n is the first argument of
this function

void
init_int(char*,char*,char*,char*,char*,char*,char*,bool,bool);
          //initializations for integer operations

void
init_bool(char*,char*,char*,char*,char*,char*,char*,bool,bool,ch
ar); //initializations for boolean operations

set<int> free_map_indices;

int get_free_map();                             //returns an index
for a map to be used in switch statements

int cur_id;                                          //keeps track
of number of temporary identifiers assigned

int cur_lab;                                    //keeps track of
number of labels assigned

map<string,string> sym_tab;                     //symbol table to
keep track of identifiers declared so far for semantic analysis
```

```
char* finalcode;                              //contains the
final code generated by concatenating all statement codes

struct stype

{

     string expcode;

     string casebeg;

     string exptype;

};

map<string,stype> smap[10];                              //At max
there can be 10 nested switches at a time and infinite parallel
switches in a code

char* switchlab;

%}

%union

{

     char* str;

     struct vartype

     {

          char* code;

          char* begin;

          char* after;

          char* place;

          char* type;

          bool terminal;

          int mindex;

          char* switchlab;
```

```
     } var;
}


%token TR FL
%token<str> NUM ID
%left EQ NEQ
%left OR LOR
%left AND LAND
%left XOR
%left NOT
%left ADD SUB
%left MUL DIV MOD
%right POW
%token INT UINT BOOL
%token ASGN ADDASGN SUBASGN MULASGN DIVASGN
%left LT LTEQ GT GTEQ
%token IF WHILE SWITCH BREAK DEFAULT CASE TERM CLN LP RP LC RC
%type<var> expr term program statement decl_stat asgn_stat
if_stat while_stat cases switch_stat
%nonassoc IFX
%nonassoc ELSE
%%


program:                                                {

     $$.code = (char*)malloc(1);
```

```
     strcpy($$.code,"");

                                                    }
          |program statement                        {

     $$.code = (char*)malloc(50000);

     strcpy($$.code,$1.code);

     strcat($$.code,$2.code);

     strcpy(finalcode,$$.code);

                                                    }
          ;
statement:
          asgn_stat
          |decl_stat
          |if_stat
          |while_stat
          |LC program RC                            {

     $$.code = (char*)malloc(1000);

     strcpy($$.code,$2.code);

                                                    }
          |switch_stat
          |TERM                                     {
```

```
     $$.code = (char*)malloc(1);

     strcpy($$.code,"");
                                                          }
          ;
switch_stat:
          SWITCH LP expr RP LC cases RC      {

     $$.code = (char*)malloc(1000);

     strcpy($$.code,"");

     if(!$3.terminal)

     strcat($$.code,$3.code);
                                                          int
ind = $6.mindex;

     map<string,stype>::iterator it = smap[ind].begin();

     string exptype = string($3.type);

     //type checking for the case expression and switch
expression

     for(;it!=smap[ind].end();it++)
                                                          {

     if(string($3.type)!=it->second.exptype)
```

```
     {

     cout<<"Incompatible types in case expression and switch
expression!"<<endl;

     exit(1);

     }

     strcat($$.code,(char*)(it->second.expcode).c_str());
                                                              }
                                                              it =
smap[ind].begin();

     //code for all the case expressions which are to be
evaluated before entering the switch

     for(;it!=smap[ind].end();it++)
                                                              {

     char* frst = (char*)malloc(10);

     strcpy(frst,(char*)(it->first).c_str());

     char* scnd = (char*)malloc(10);

     strcpy(scnd,(char*)(it->second.casebeg).c_str());

     concat(8,$$.code,"if ",$3.place,"=",frst," goto
",scnd,"\n");
```

```
                                                                    }

     concat(4,$$.code,$6.code,$6.switchlab,":\n");

     smap[ind].clear();

     free_map_indices.insert(ind);

                                                                  }

          ;

cases:
          CASE expr CLN statement BREAK TERM cases     {

     //this production is used when a break follows the case

     $$.code = (char*)malloc(1000);

     $$.begin = (char*)malloc(10);

     $$.begin = getNewLabel($$.begin);

     $$.type = (char*)malloc(5);

     int ind = $7.mindex;

     smap[ind][string($2.place)].casebeg = string($$.begin);

     smap[ind][string($2.place)].exptype = string($2.type);

     if($2.terminal)
```

```
     {

          smap[ind][string($2.place)].expcode = "";

     }

     else

          smap[ind][string($2.place)].expcode = string($2.code);

     strcpy($$.code,"");

     concat(8,$$.code,$$.begin,":\n",$4.code,"goto
",$7.switchlab,"\n",$7.code);

     $$.mindex = ind;

     $$.switchlab = (char*)malloc(10);

     strcpy($$.switchlab,$7.switchlab);

     }
          |CASE expr CLN statement cases          {

     //this production is used when a break does not follow the
case

     $$.code = (char*)malloc(1000);

     $$.begin = (char*)malloc(10);
```

```
     $$.begin = getNewLabel($$.begin);

     $$.type = (char*)malloc(5);
                                                              int
ind = $5.mindex;

     smap[ind][string($2.place)].casebeg = string($$.begin);

     smap[ind][string($2.place)].exptype = string($2.type);

     if($2.terminal)
                                                              {

     smap[ind][string($2.place)].expcode = "";
                                                              }
                                                              else

     smap[ind][string($2.place)].expcode = string($2.code);

     strcpy($$.code,"");

     concat(5,$$.code,$$.begin,";\n",$4.code,$5.code);

     $$.mindex = ind;

     $$.switchlab = (char*)malloc(10);

     strcpy($$.switchlab,$5.switchlab);
                                                         }
```

```
        |DEFAULT CLN statement                    {
    $$.code = (char*)malloc(1000);
    strcpy($$.code,$3.code);
    $$.mindex = get_free_map();
    $$.switchlab = (char*)malloc(10);
    strcpy($$.switchlab,getNewLabel($$.switchlab));
                                                    }
        |                                           {
    $$.code = (char*)malloc(1);
    strcpy($$.code,"");
    $$.mindex = get_free_map();
    $$.switchlab = (char*)malloc(10);
    strcpy($$.switchlab,getNewLabel($$.switchlab));
                                                    }
        ;
while_stat:
        WHILE LP expr RP statement                  {
```

```
     //only boolean expressions are allowed within the
paranthesis

     if(strcmp($3.type,"bool")!=0)

     {

     cout<<"Incompatible type in while condition!"<<endl;

     exit(1);

     }

     $$.code = (char*)malloc(500);

     strcpy($$.code,"");


     if($3.begin==NULL)

     {

     $3.begin = (char*)malloc(10);

     $3.begin = getNewLabel($3.begin);

     }

     strcat($$.code,$3.begin);

     strcat($$.code,":\n");
```

```
     if(!$3.terminal)

     {

     strcat($$.code,$3.code);

     }

     concat(4,$$.code,"if ",$3.place,"==false goto ");

     if($5.after==NULL)

     {

     $5.after = (char*)malloc(10);

     $5.after = getNewLabel($5.after);

     }

     concat(9,$$.code,$5.after,"\n",$5.code,"goto
",$3.begin,"\n",$5.after,":\n");
                                                                  }
          ;
if_stat:
          IF LP expr RP statement       %prec IFX {

     //only boolean expressions are allowed within the
paranthesis
```

```
if(strcmp($3.type,"bool")!=0)

{

cout<<"Incompatible type in if condition!"<<endl;

exit(1);

}

$$.code = (char*)malloc(500);

strcpy($$.code,"");

if(!$3.terminal)

strcat($$.code,$3.code);

concat(4,$$.code,"if ",$3.place,"==false goto ");

if($5.after==NULL)

{

$5.after = (char*)malloc(10);

$5.after = getNewLabel($5.after);

}
```

```
concat(6,$$.code,$5.after,"\n",$5.code,$5.after,":\n");
                                                        }
     |IF LP expr RP statement ELSE statement {
if(strcmp($3.type,"bool")!=0)
{
cout<<"Incompatible type in if condition!"<<endl;
exit(1);
}
$$.code = (char*)malloc(500);
strcpy($$.code,"");
if(!$3.terminal)
strcat($$.code,$3.code);
concat(4,$$.code,"if ",$3.place,"==false goto ");
if($7.begin==NULL)
{
$7.begin = (char*)malloc(10);
```

```
     $7.begin = getNewLabel($7.begin);

     }

     concat(4,$$.code,$7.begin,"\n",$5.code);

     if($7.after==NULL)

     {

     $7.after = (char*)malloc(10);

     $7.after = getNewLabel($7.after);

     }

     concat(9,$$.code,"goto
",$7.after,"\n",$7.begin,":\n",$7.code,$7.after,":\n");
                                                                  }
          ;
decl_stat:
          INT ID TERM                                        {

     $$.code = (char*)malloc(50);

     strcpy($$.code,"assign ");

     strcat($$.code,$2);

     strcat($$.code," 4 bytes\n");
```

```
sym_tab[$2] = "int";
                                                    }
     |UINT ID TERM                              {
$$.code = (char*)malloc(50);
strcpy($$.code,"assign ");
strcat($$.code,$2);
strcat($$.code," 4 bytes\n");
sym_tab[$2] = "uint";
                                                    }
     |BOOL ID TERM                              {
$$.code = (char*)malloc(50);
strcpy($$.code,"assign ");
strcat($$.code,$2);
strcat($$.code," 1 byte\n");
sym_tab[$2] = "bool";
                                                    }
     ;
```

```
asgn_stat:
          INT ID ASGN expr TERM                          {

     //uint variables can be assigned to int variables

     if(strcmp($4.type,"bool")==0)

     {

     cout<<"Incompatible types!"<<endl;

     exit(1);

     }

     $$.code = (char*)malloc(500);

     strcpy($$.code,"assign ");

     strcat($$.code,$2);

     strcat($$.code," 4 bytes\n");

     if(!$4.terminal)

     {

     strcat($$.code,$4.code);

     }
```

```
concat(4,$$.code,$2,"=",$4.place);

sym_tab[$2] = "int";

strcat($$.code,"\n");
                                                        }

     |UINT ID ASGN expr TERM                        {

if(sym_tab.find($4.place)==sym_tab.end())

{

if(strcmp($4.type,"bool")==0)

{

     cout<<"Incompatible types!"<<endl;

     exit(1);

}

}

else

{

if(strcmp($4.type,"uint")!=0)
```

```
{
     cout<<"Incompatible types!"<<endl;
     exit(1);
}
}
$$.code = (char*)malloc(500);
strcpy($$.code,"assign ");
strcat($$.code,$2);
strcat($$.code," 4 bytes\n");
if(!$4.terminal)
{
strcat($$.code,$4.code);
}
concat(4,$$.code,$2,"=",$4.place);
sym_tab[$2] = "uint";
```

```
strcat($$.code,"\n");
                                                                }

     |BOOL ID ASGN expr TERM                          {

//boolean identifier will accept only boolean expression

if(strcmp($4.type,"bool")!=0)

{

cout<<"Incompatible types!"<<endl;

exit(1);

}

$$.code = (char*)malloc(500);

strcpy($$.code,"assign ");

strcat($$.code,$2);

strcat($$.code," 1 byte\n");

if(!$4.terminal)

{

strcat($$.code,$4.code);
```

```
}
concat(4,$$.code,$2,"=",$4.place);
sym_tab[$2] = "bool";
strcat($$.code,"\n");
                                                    }
     |ID ASGN expr TERM                              {
//assign a value to an already declared identifier
if(sym_tab.find($1)==sym_tab.end())
{
cout<<"Undefined Symbol: "<<$1;
exit(1);
}
if(sym_tab[$1]!=string($3.type))
{
if(sym_tab[$1]=="bool" || strcmp($3.type,"bool")==0)
{
```

```
     cout<<"Incompatible types!"<<endl;
     exit(1);
}
}
$$.code = (char*)malloc(500);
strcpy($$.code,"");
if(!$3.terminal)
{
strcat($$.code,$3.code);
}
concat(5,$$.code,$1,"=",$3.place,"\n");
                                                    }
     |ID ADDASGN expr TERM                          {
//short hand notation for addition along with assignment
if(sym_tab.find($1)==sym_tab.end())
{
```

```
cout<<"Undefined Symbol: "<<$1;

exit(1);

}

if(sym_tab[$1]!=string($3.type))

{

if(sym_tab[$1]=="bool" || strcmp($3.type,"bool")==0)

{

     cout<<"Incompatible types!"<<endl;

     exit(1);

}

}

$$.code = (char*)malloc(500);

strcpy($$.code,"");

if(!$3.terminal)

{
```

```
strcat($$.code,$3.code);

}

concat(7,$$.code,$1,"=",$1,"+",$3.place,"\n");
                                                    }
     |ID SUBASGN expr TERM                          {

//short hand notation for subtraction along with assignment

if(sym_tab.find($1)==sym_tab.end())

{

cout<<"Undefined Symbol: "<<$1;

exit(1);

}


if(sym_tab[$1]!=string($3.type))

{

if(sym_tab[$1]=="bool" || strcmp($3.type,"bool")==0)

{

     cout<<"Incompatible types!"<<endl;
```

```
          exit(1);

     }

     }

     $$.code = (char*)malloc(500);

     strcpy($$.code,"");

     if(!$3.terminal)

     {

     strcat($$.code,$3.code);

     }

     concat(7,$$.code,$1,"=",$1,"-",$3.place,"\n");
                                                                    }
          |ID MULASGN expr TERM                          {

     //short hand notation for multiplication along with
assignment

     if(sym_tab.find($1)==sym_tab.end())

     {

     cout<<"Undefined Symbol: "<<$1;
```

```
exit(1);

}

if(sym_tab[$1]!=string($3.type))

{

if(sym_tab[$1]=="bool" || strcmp($3.type,"bool")==0)

{

     cout<<"Incompatible types!"<<endl;

     exit(1);

}

}

$$.code = (char*)malloc(500);

strcpy($$.code,"");

if(!$3.terminal)

{

strcat($$.code,$3.code);
```

```
}
concat(7,$$.code,$1,"=",$1,"*",$3.place,"\n");
                                                    }
     |ID DIVASGN expr TERM                          {
//short hand notation for division along with assignment
if(sym_tab.find($1)==sym_tab.end())
{
cout<<"Undefined Symbol: "<<$1;
exit(1);
}
if(sym_tab[$1]!=string($3.type))
{
if(sym_tab[$1]=="bool" || strcmp($3.type,"bool")==0)
{
     cout<<"Incompatible types!"<<endl;
     exit(1);
```

```
    }

    }

    $$.code = (char*)malloc(500);

    strcpy($$.code,"");

    if(!$3.terminal)

    {

    strcat($$.code,$3.code);

    }

    concat(7,$$.code,$1,"=",$1,"/",$3.place,"\n");
                                                                    }



        ;
expr:
        term                                {

    //Terminal symbol, either a number, an identifier, true or
false value
                                                                $$.code =
(char*)malloc(50);
```

```
     strcpy($$.code,$1.place);

     strcpy($$.type,$1.type);

     $$.terminal = true;
                                                  }
          |expr ADD expr                     {
                                                       //Add two
expressions with type checking
                                                       $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                       $$.place
= (char*)malloc(10);
                                                       $$.type =
(char*)malloc(5);

     init_int($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,$
$.type,$1.terminal,$3.terminal);

     concat(7,$$.code,$$.place,"=",$1.place,"+",$3.place,"\n");

     $$.terminal = false;
                                                  }
          |expr SUB expr                     {

     //Subtract two expressions with type checking

                                                       $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                       $$.place
= (char*)malloc(10);
```

```
                                                    $$.type =
(char*)malloc(5);

     init_int($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,$
$.type,$1.terminal,$3.terminal);

     concat(7,$$.code,$$.place,"=",$1.place,"-",$3.place,"\n");

     $$.terminal = false;
                                                 }
          |expr MUL expr                    {

     //Multiply two expressions with type checking
                                                    $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                    $$.place
= (char*)malloc(10);
                                                    $$.type =
(char*)malloc(5);

     init_int($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,$
$.type,$1.terminal,$3.terminal);

     concat(7,$$.code,$$.place,"=",$1.place,"*",$3.place,"\n");

     $$.terminal = false;
                                                 }
          |expr DIV expr                    {
                                                    //Divide
two expressions with type checking
```

```
                                                    $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                    $$.place
= (char*)malloc(10);
                                                    $$.type =
(char*)malloc(5);

     init_int($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,$
$.type,$1.terminal,$3.terminal);

     concat(7,$$.code,$$.place,"=",$1.place,"/",$3.place,"\n");

     $$.terminal = false;
                                                  }
          |expr MOD expr                      {
                                                    //Modular
divide two expressions with type checking
                                                    $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                    $$.place
= (char*)malloc(10);
                                                    $$.type =
(char*)malloc(5);

     init_int($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,$
$.type,$1.terminal,$3.terminal);

     concat(7,$$.code,$$.place,"=",$1.place,"%",$3.place,"\n");

     $$.terminal = false;
                                                  }
```

```
          |expr POW expr                    {

     //Exponentiation (right associative)

                                                  $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);

                                                  $$.place
= (char*)malloc(10);

                                                  $$.type =
(char*)malloc(5);

     init_int($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,$
$.type,$1.terminal,$3.terminal);

     concat(7,$$.code,$$.place,"=",$1.place,"@",$3.place,"\n");

     $$.terminal = false;

                                                }
          |expr AND expr                    {

                                                  //Bitwise
AND on two integers / unsigned integers

                                                  $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);

                                                  $$.place
= (char*)malloc(10);

                                                  $$.type =
(char*)malloc(5);

     init_int($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,$
$.type,$1.terminal,$3.terminal);

     concat(7,$$.code,$$.place,"=",$1.place,"&",$3.place,"\n");
```

```
     $$.terminal = false;
                                                    }
          |expr OR expr                        {
                                                         //Bitwise
OR on two integers / unsigned integers
                                                         $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                         $$.place
= (char*)malloc(10);
                                                         $$.type =
(char*)malloc(5);

     init_int($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,$
$.type,$1.terminal,$3.terminal);

     concat(7,$$.code,$$.place,"=",$1.place,"|",$3.place,"\n");

     $$.terminal = false;
                                                    }
          |expr XOR expr                       {
                                                         //Bitwise
XOR on two integers / unsigned integers
                                                         $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                         $$.place
= (char*)malloc(10);
                                                         $$.type =
(char*)malloc(5);
```

```
     init_int($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,$
$.type,$1.terminal,$3.terminal);

     concat(7,$$.code,$$.place,"=",$1.place,"^",$3.place,"\n");

     $$.terminal = false;
                                                  }
          |NOT expr                          {
                                                        $$.code =
(char*)malloc(strlen($2.code)+20);
                                                        $$.place
= (char*)malloc(10);
                                                        $$.place
= getNewId($$.place);

     sym_tab[$$.place] = sym_tab[$2.place];
                                                        $$.type =
(char*)malloc(5);

     strcpy($$.type,(char*)sym_tab[$2.place].c_str());

     strcpy($$.code,"");

     if(!$$.terminal)
                                                        {

     strcat($$.code,$2.code);
                                                        }

     concat(6,$$.code,$$.place,"=","~",$2.place,"\n");
```

```
     $$.terminal = false;
                                                    }
          |expr LAND expr                           {
                                                         //Boolean
AND on boolean variables
                                                         $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                         $$.place
= (char*)malloc(10);
                                                         $$.type =
(char*)malloc(5);

     init_bool($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,
$$.type,$1.terminal,$3.terminal,'b');

     concat(7,$$.code,$$.place,"=",$1.place,"&&",$3.place,"\n");

     $$.terminal = false;
                                                    }
          |expr LOR expr                       {
                                                         //Boolean
OR on boolean variables
                                                         $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                         $$.place
= (char*)malloc(10);
                                                         $$.type =
(char*)malloc(5);
```

```
     init_bool($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,
$$.type,$1.terminal,$3.terminal,'b');

     concat(7,$$.code,$$.place,"=",$1.place,"||",$3.place,"\n");

     $$.terminal = false;
                                                  }

          |expr EQ expr                       {
                                                       //Check
equality of two expressions
                                                       $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                       $$.place
= (char*)malloc(10);
                                                       $$.type =
(char*)malloc(5);

     init_bool($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,
$$.type,$1.terminal,$3.terminal,'e');

     concat(7,$$.code,$$.place,"=",$1.place,"==",$3.place,"\n");

     $$.terminal = false;
                                                  }
          |expr NEQ expr                      {
                                                       //Check
inequality of two expressions
                                                       $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
```

```
                                                        $$.place
= (char*)malloc(10);
                                                        $$.type =
(char*)malloc(5);

     init_bool($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,
$$.type,$1.terminal,$3.terminal,'e');

     concat(7,$$.code,$$.place,"=",$1.place,"!=",$3.place,"\n");

     $$.terminal = false;
                                                    }
          |expr LT expr                        {
                                                        //Check
if first expression has a lesser value than the second
expression
                                                        $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                        $$.place
= (char*)malloc(10);
                                                        $$.type =
(char*)malloc(5);

     init_bool($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,
$$.type,$1.terminal,$3.terminal,'i');

     concat(7,$$.code,$$.place,"=",$1.place,"<",$3.place,"\n");

     $$.terminal = false;
                                                    }
          |expr LTEQ expr                          {
```

```
                                                        //Check
if first expression has a lesser or equal value than the second
expression

                                                        $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);

                                                        $$.place
= (char*)malloc(10);

                                                        $$.type =
(char*)malloc(5);


     init_bool($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,
$$.type,$1.terminal,$3.terminal,'i');


     concat(7,$$.code,$$.place,"=",$1.place,"<=",$3.place,"\n");


     $$.terminal = false;

                                                    }

          |expr GT expr                        {

                                                        //Check
if first expression has a greater value than the second
expression

                                                        $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);

                                                        $$.place
= (char*)malloc(10);

                                                        $$.type =
(char*)malloc(5);


     init_bool($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,
$$.type,$1.terminal,$3.terminal,'i');


     concat(7,$$.code,$$.place,"=",$1.place,">",$3.place,"\n");
```

```
     $$.terminal = false;
                                                    }
          |expr GTEQ expr                           {
                                                         //Check
if first expression has a greater or equal value than the second
expression
                                                         $$.code =
(char*)malloc(strlen($1.code)+strlen($3.code)+25);
                                                         $$.place
= (char*)malloc(10);
                                                         $$.type =
(char*)malloc(5);

     init_bool($1.type,$3.type,$$.code,$1.code,$3.code,$$.place,
$$.type,$1.terminal,$3.terminal,'i');

     concat(7,$$.code,$$.place,"=",$1.place,">=",$3.place,"\n");

     $$.terminal = false;
                                                    }
          |LP expr RP                               {

     //Parenthesize the expression
                                                         $$.code =
(char*)malloc(strlen($2.code)+10);

     strcpy($$.code,$2.code);

     $$.terminal = false;
```

```
                                                    $$.type =
(char*)malloc(5);

     strcpy($$.type,$2.type);
                                                    $$.place
= (char*)malloc(10);

     strcpy($$.place,$2.place);
                                                }
          ;
term:
          NUM                                   {
                                                    $$.place
= (char*)malloc(10);
                                                    $$.type =
(char*)malloc(5);

     strcpy($$.type,"int");

     strcpy($$.place,$1);
                                                }
          |TR                                   {
                                                    $$.place
= (char*)malloc(10);
                                                    $$.type =
(char*)malloc(5);

     strcpy($$.type,"bool");

     strcpy($$.place,"true");
```

```
                                                    }
          |FL                                       {
                                                         $$.place
= (char*)malloc(10);
                                                         $$.type =
(char*)malloc(5);
     strcpy($$.type,"bool");
     strcpy($$.place,"false");
                                                    }
          |ID                                       {
     if(sym_tab.find($1)==sym_tab.end())
                                                         {
     cout<<"Undefined symbol: "<<$1<<endl;
     exit(1);
                                                         }
                                                         $$.place
= (char*)malloc(20);
                                                         $$.type =
(char*)malloc(5);
     strcpy($$.type,(char*)sym_tab[$1].c_str());
     strcpy($$.place,$1);
                                                    }
```

```
          ;

%%

int main(int argc, char **argv)
{
     cur_id = 1;
     cur_lab = 1;
     finalcode = (char*)malloc(50000);
     for(int i=0;i<10;i++)
          free_map_indices.insert(i);
     yyparse();
     cout<<finalcode<<endl;
     return 0;
}

void itoa(char* ret,int n)
{
     int i=0;
     while(n)
     {
          ret[i++] = (n%10+'0');
          n/=10;
     }
     for(int j=0;j<i/2;j++)
```

```
     {
          char tmp = ret[j];
          ret[j] = ret[i-j-1];
          ret[i-j-1] = tmp;
     }
     ret[i] = '\0';
}

char* getNewId(char* id)
{
     strcpy(id,"_t");
     itoa(id+2,cur_id);
     cur_id++;
     return id;
}

char* getNewLabel(char* id)
{
     strcpy(id,"_L");
     itoa(id+2,cur_lab);
     cur_lab++;
     return id;
}

int get_free_map()
```

```
{
     int index = *free_map_indices.begin();
     free_map_indices.erase(index);
     return index;
}


void concat(int size,...)
{
     va_list varlist;
     va_start(varlist,size);
     char* target = va_arg(varlist,char*);
     for(int i=1;i<size;i++)
     {
          strcat(target,va_arg(varlist,char*));
     }
}


void
init_int(char*t1,char*t2,char*code,char*c1,char*c2,char*place,ch
ar*type,bool term1,bool term2)
{
     if(strcmp(t1,"bool")==0 || strcmp(t2,"bool")==0)
     {
          cout<<"Incompatible types (init_int)!"<<endl;
          exit(1);
     }
```

```
     place = getNewId(place);
     sym_tab[place] = "int";
     strcpy(type,"int");
     strcpy(code,"");
     if(!term1)
     {
          strcat(code,c1);
     }
     if(!term2)
     {
          strcat(code,c2);
     }
}


void
init_bool(char*t1,char*t2,char*code,char*c1,char*c2,char*place,c
har*type,bool term1,bool term2,char t)
{
     if(t=='b')
     {
          if(strcmp(t1,"bool")!=0 || strcmp(t2,"bool")!=0)
          {
               cout<<"Incompatible types (init_bool)!"<<endl;
               exit(1);
          }
     }
```

```
     else if(t=='i')
     {
          if(strcmp(t1,"bool")==0 || strcmp(t2,"bool")==0)
          {
               cout<<"Incompatible types (init_bool)!"<<endl;
               exit(1);
          }
     }
     place = getNewId(place);
     sym_tab[place] = "bool";
     strcpy(type,"bool");
     strcpy(code,"");
     if(!term1)
     {
          strcat(code,c1);
     }
     if(!term2)
     {
          strcat(code,c2);
     }
}

void yyerror(const char *s)
{
     fprintf(stderr, "error: %s\n", s);
```

```
}
```

SAMPLE INPUT AND OUTPUT FOR TESTING

INPUT FILE

The following Sub-C code was used to test the project. This code involves the use of if-else statements, while statement, nested switches, declaration of all allowed types of variables and evaluation of expressions with most of the allowed operators.

```
int a = 15;
uint b = 6;
bool c = (((a|b)&(5+10*9-6/2))@(4*3)) < 50;
bool d;
if(c==true)
{
     while(b<15)
     {
          if(a%2==0)
          {
               a = a+1;
               b = b+2;
          }
          else
          {
               b = b+1;
          }
     }
     a = 100;
     b = 100;
}
```

```
else
{
     switch(a)
     {
          case 10:
```

OUTPUT GENERATED

```
assign a 4 bytes
a=15
assign b 4 bytes
b=6
assign c 1 byte
_t1=a|b
_t2=10*9
_t3=5+_t2
_t4=6/2
_t5=_t3-_t4
_t6=_t1&_t5
_t7=4*3
_t8=_t6@_t7
_t9=_t8<50
c=_t9
assign d 1 byte
_t10=c==true
if _t10==false goto _L10
_L3:
_t11=b<15
if _t11==false goto _L4
_t12=a%2
_t13=_t12==0
if _t13==false goto _L1
_t14=a+1
a=_t14
_t15=b+2
b=_t15
goto _L2
_L1:
_t16=b+1
b=_t16
_L2:
goto _L3
_L4:
a=100
b=100
goto _L11
```

```
_L10:
if a=10 goto _L9
if a=15 goto _L8
_L9:
a=1
goto _L7
_L8:
_L5:
_t17=a!=0
```

MAJOR LIMITATIONS AND ASSUMPTIONS

- The language supports only while loops in repetitive constructs.

- The break statement can be used only in switch statements.
- The language supports only three data types: integer, unsigned integer and boolean.
- An integer variable can store unsigned integer variable but not vice versa.
- A bool variable can be used only with another bool variable in an expression.
- The expressions in if and while conditions need to be of bool type.
- The else block is matched with the closest if.