Compiler Design

Project Report - 4

Intermediate Code Generator

By:

15CO248 - Jeshventh Raja T K

15CO229 - Zoeb Mithaiwala

CONTENTS

Introduction

What is a compiler?

A compiler is a special program that processes statements written in a high level programming language and turns them into machine language.

The input given to the compiler is a source code written in a high level language. The compiler outputs an equivalent and correct target code in machine level language. The compiler checks and outputs errors, if present. The compiler can check for syntax and semantic errors only.

There are two phases in compiling:

1) Analysis phase (Front end)
2) Synthesis phase (Back end)

Analysis phase

This phase consists of four steps:

1) Lexical analysis - Read and convert characters into words. Done by lexical/linear analyser or scanner
2) Syntax analysis - Convert the words into sentences. Done by syntax analyser or parser
3) Semantic analysis - Understand the meaning of sentences.
4) Intermediate code generation - Convert the understanding into intermediate code

Synthesis phase

This phase consists of two steps:

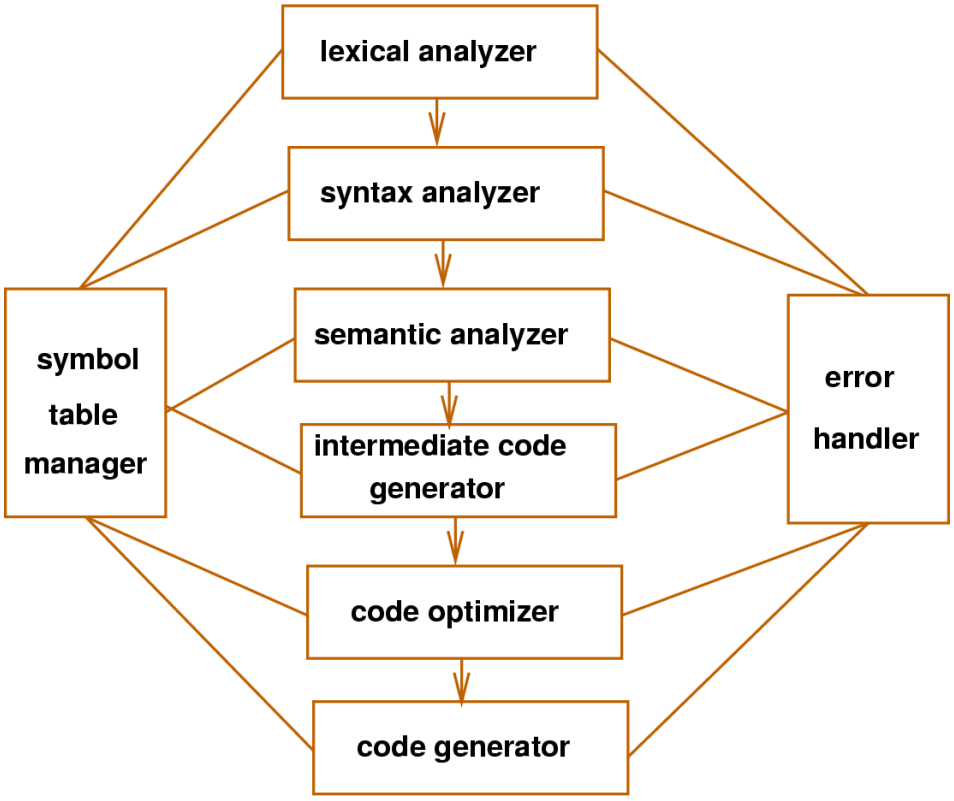
1) Code optimisation - Remove any unwanted or redundant code
2) Code generation - Convert to machine level language

There are various types of compilers:

1) Retargetable compiler - Converts H/L language to M/L language that is understood by a different machine
2) Single pass compiler - It scans an instruction, converts it to machine level code and then goes to the next instruction
3) Multipass compiler - It processes the source code or abstract syntax tree of a program several times, generating an output file after each pass

Additional functionalities of the compiler can be:

1) Error handling
2) Fast compilation
3) Efficient machine level code
4) Fast executing machine level output



Intermediate Code Generation

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

The benefits of using machine independent intermediate code are:

1. Because of the machine independent intermediate code, portability will be enhanced.For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
2. Retargeting is facilitated
3. It is easier to apply source code modification to improve the performance of source code by optimising the intermediate code.

Backpatching

There are times when the compiler has to execute a jump instruction but it doesn't know where to yet. So it will fill in a blank value at this point and remember that this happened. Then once the value is found that will actually be used the compiler will go back "backpatch" and fill in the correct value.

Code - LEX

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    //#include "y.tab.h"
    int lineNo=1;
%}

keyword
auto|break|case|char|const|continue|default|do|double|else|enum|extern|f
loat|for|goto|if|int|long|register|return|short|signed|sizeof|static|str
uct|switch|typedef|union|unsigned|void|volatile|while
singleLineComment \/\/.*
multilineComment "/*"([^*]|\*+[^*/])*\*+"/"
multilineCommentError "/*"([^*]|\*+[^*/])*
str \"[^\"]*\"
strError \"[^\"]*
letter [a-zA-Z]
digit [0-9]
nonIdentifier ({digit})+(_|{letter})(_|{letter}|{digit})*
identifier (_|{letter})(_|{letter}|{digit})*
dataType int|float|double|short\ int|long\ int
argument {dataType}[\ ]*{identifier}
integer ({digit})+
float ({digit})*\.({digit})+
assignmentOperator "="
arithmeticOperator "+"|"-"|"*"|"/"|"%"
operator
"+="|"-="|"*="|"/="|"%="|">>="|"<<="|"&="|"^="|"|="|"++"|"--"|"=="|"!="|
">"|"<"|">="|"<="|"||"|"&&"|"!"|"&"|"|"|"^"|"~"|"<<"|">>"|"?"
headerFile "#include"[\ ]*"<"[\ ]*[^>]*[\ ]*">"
specialCharacters \;|\{|\}|\(|\)|\[|\],
whitespaces [\ ]|[\t]
%%
{multilineCommentError} {printf("ERROR: Multiline comment is not
closed\n");}
{strError} {printf("ERROR: String is not closed\n");}
{singleLineComment} {}
{multilineComment} {}
{headerFile} {return HEADERFILE;}
{str} {
       char *te = (char *)malloc(strlen(yytext)-1);
```

```
       int l = strlen(yytext);
       strncpy(te, yytext + 1, l-2);
         yylval.sym=te; return STR;}
auto {  yylval.sym="auto"; return AUTO;}
break {  yylval.sym="break"; return BREAK;}
case {  yylval.sym="case"; return CASE;}
char {  yylval.sym="keyword"; return CHAR;}
const {  yylval.sym="const"; return CONST;}
continue {  yylval.sym="continue"; yylval.sym=""; return CONTINUE;}
default {  yylval.sym="default"; return DEFAULT;}
do {  yylval.sym="do"; return DO;}
double {  yylval.sym="double"; return DOUBLE;}
else {  yylval.sym="else"; return ELSE;}
enum {  yylval.sym="enum"; return ENUM;}
extern {  yylval.sym="extern"; return EXTERN;}
float {  yylval.sym="float"; return FLOAT;}
for {  yylval.sym="for"; return FOR;}
goto {  yylval.sym="goto"; return GOTO;}
if {  yylval.sym="if"; return IF;}
int { yylval.sym="int"; return INT;}
long {  yylval.sym="long"; return LONG;}
register {  yylval.sym="register"; return REGISTER;}
return {  yylval.sym="return"; return RETURN;}
short {  yylval.sym="short"; return SHORT;}
signed {  yylval.sym="signed"; return SIGNED;}
sizeof {  yylval.sym="sizeof"; return SIZEOF;}
static {  yylval.sym="static"; return STATIC;}
struct {  yylval.sym="struct"; return STRUCT;}
switch {  yylval.sym="switch"; return SWITCH;}
typedef {  yylval.sym="typedef"; return TYPEDEF;}
union {  yylval.sym="union"; return UNION;}
unsigned {  yylval.sym="unsigned"; return UNSIGNED;}
void {  yylval.sym="void"; return VOID;}
volatile {  yylval.sym=yytext; return VOLATILE;}
while {  yylval.sym="while"; return WHILE;}
{identifier} {  yylval.sym=yytext; return ID;}
{integer} {  yylval.sym=yytext; return CONSTANT;}
{float} {  yylval.sym=yytext; return CONSTANT;}
{nonIdentifier} {printf("ERROR: %s is an ill formed token\n", yytext);}

"+="  {  return ADD_ASSIGN;}
"-="  {  return SUB_ASSIGN;}
"*="  {  return MUL_ASSIGN;}
"/="  {  return DIV_ASSIGN;}
"%="  {  return MOD_ASSIGN;}
```

```
">>=" {  return LEFT_ASSIGN;}
"<<=" {  return RIGHT_ASSIGN;}
"&="  {  return AND_ASSIGN;}
"^="  {  return XOR_ASSIGN;}
"|="  {  return OR_ASSIGN;}
"++"  {  return INC;}
"--"  {  return DEC;}
"=="  {  return EQ;}
"!="  {  return NE;}
">"   {  return '>';}
"<"   {  return '<';}
">="  {  return GE;}
"<="  {  return LE;}
"||"  {  return OR;}
"&&"  {  return AND;}
"!"   {  return '!';}
"&"   {  return '&';}
"|"   {  return '|';}
"^"   {  return '^';}
"~"   {  return '~';}
"<<"  {  return LEFT;}
">>"  {  return RIGHT;}
"?"   {  return '?';}
{assignmentOperator} {  return '=';}
"-"   {  return '-';}
"+"   {  return '+';}
"*"   {  return '*';}
"/"   {  return '/';}
"%"   {  return '%';}
";"   {  return ‘;’;}
"{"   {  return '{';}
"}"   {  return '}';}
"("   {  return '(';}
")"   {  return ')';}
"["   {  return '[';}
"]"   {  return ']';}
","   {  return ',';}
{whitespaces} {}
. {printf("Error: %s is an Illegal characters\n",yytext);}
"\n" {lineNo++;}
%%
```

Code - PARSER

```
%token ID CONSTANT STR HEADERFILE

%token INC DEC LE GE EQ NE
%token AND OR LEFT RIGHT MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN

%token AUTO BREAK CASE CHAR CONST CONTINUE DEFAULT DO DOUBLE ELSE ENUM
EXTERN FLOAT FOR GOTO IF INT LONG

%token REGISTER RETURN SHORT SIGNED SIZEOF STATIC STRUCT SWITCH TYPEDEF
UNION UNSIGNED VOID VOLATILE WHILE

%union{
    int integer;
    float floating_point;
    char *sym;
    char *val;
}

%start startSymbol

%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <limits.h>

    int size=20;
    char *alc=0;
    int globalIndex=0;
    int globalScope=1;
    int newScope=1;
    char *alc1=0;
    char *alc2=0;
    char *alc3=0;
    int setDatatype=1;
    int flag=0;
    int flag2=0;
    int procFlag=-1;
    int parameterNumber=0;
```

```
    int currentScope=1;
    int rhs=0;
    int whileStart=0;
    #define ANSI_COLOR_RED    "\x1b[31m"
    #define ANSI_COLOR_RESET   "\x1b[0m"

    int yyerror(char *msg);
    int yylex();

    struct Stack
    {
       int top;
       unsigned capacity;
       int* array;
    };

    struct Stack* createStack(unsigned capacity)
    {
      struct Stack* stack = (struct Stack*) malloc(sizeof(struct
Stack));
      stack->capacity = capacity;
      stack->top = -1;
      stack->array = (int*) malloc(stack->capacity * sizeof(int));
      return stack;
    }

    int isFull(struct Stack* stack)
    {   return stack->top == stack->capacity - 1; }

    int isEmpty(struct Stack* stack)
    {   return stack->top == -1;  }

    void push(struct Stack* stack, int item)
    {
      if (isFull(stack))
            return;
      stack->array[++stack->top] = item;
    }

    int pop(struct Stack* stack)
    {
      if (isEmpty(stack))
            return INT_MIN;
      return stack->array[stack->top--];
    }
```

```
struct symbolTable{
   int index;
   char *symbol;
   char *attribute;
   char *data;
   struct symbolTable *next;
   int scope;
   int procedureDefinitionFlag;
   struct parameterChain *params;
   int nesting;
};

struct parameterChain{
   char *name;
   char *dataType;
   struct parameterChain *next;
};

struct symbolTable *hash[2][20];
struct Stack* stack;
struct Stack* stack1;
int i=0;

void init()
{
     stack = createStack(100);
     alc2 = (char*)malloc(sizeof(char)*100);
     alc3 = (char*)malloc(sizeof(char)*100);
     stack1 = createStack(100);
   for(;i<size;i++)
   {
         hash[0][i] = NULL;
         hash[1][i] = NULL;
   }
}

int hashLocation(char *sym)
{
   int sum=0;
   size_t length = strlen(sym);
   int k=0;
   for(;k<length;k++)
   {
         sum = sum + (int)sym[k];
```

```
        }
        return sum%size;
    }

    int searchHash(int type, char *sym, int x)
    {
        struct symbolTable *temp = hash[type][x];
        while(temp!=NULL)
        {
              if (strcmp(temp->symbol, sym)==0 && globalScope ==
temp->scope)
              {
                    return 0;
              }
              temp=temp->next;
        }
        return 1;
    }

    int searchMain()
    {
        struct symbolTable *temp = hash[0][421%size];
        while(temp!=NULL)
        {
              if (strcmp(temp->symbol, "main")==0 && temp->scope==0)
              {
                    return 1;
              }
              temp = temp->next;
        }
        return 0;
    }

    int getDatatype(char *sym)
    {
        int x = hashLocation(sym);
        int i=stack->top;
        for(;i>=0;i--)
        {
              struct symbolTable *temp = hash[0][x];
              while(temp!=NULL)
              {
              if(strcmp(temp->symbol, sym)==0 && temp->scope ==
stack->array[i])
              {
```

```
                    if (strcmp(temp->data, "float")==0)
                    return 2;
                    else if (strcmp(temp->data, "int")==0)
                    return 1;
                    else if(strcmp(temp->data, "void")==0)
                    return 0;
              }
              temp = temp->next;
              }
       }
       return -1;
    }

    void addToTable(int type, char *sym, char *attr, char *dat)
    {
       int x = hashLocation(sym);
       if (searchHash(type, sym, x)==0)
             return;
       struct symbolTable *newSymbol = (struct symbolTable
*)malloc(sizeof(struct symbolTable));
       char *te = (char *)malloc(strlen(sym)+1);
       strcpy(te, sym);
       newSymbol->symbol = te;
       newSymbol->attribute = attr;
       newSymbol->data = dat;
       newSymbol->next = NULL;
       newSymbol->index = globalIndex + 1;
       newSymbol->nesting = currentScope - 1;
       newSymbol->scope = globalScope - 1;
       newSymbol->params = NULL;
       newSymbol->procedureDefinitionFlag = procFlag;
       globalIndex++;
       struct symbolTable *temp = hash[type][x];
       hash[type][x] = newSymbol;
       hash[type][x]->next = temp;
    }

    void addParams(char *dataType, char *sym)
    {
       int x = hashLocation(alc2);
       struct symbolTable *temp = hash[0][x];
       while(strcmp(temp->symbol, alc2)!=0 )
       {
             temp=temp->next;
             if(temp==NULL)
```

```
            {
                  printf("\nTemp = null when searching for function
name in symbol table\n");
                  return;
            }
      }
      struct parameterChain *newParam = (struct
parameterChain*)malloc(sizeof(struct parameterChain));
      char *te = (char *)malloc(strlen(sym)+1);
      strcpy(te, sym);
      newParam->name = te;
      newParam->dataType = dataType;
      newParam->next=NULL;
      struct parameterChain *temp1 = temp->params;
      if(temp1==NULL)
      {
            temp->params=newParam;
            return;
      }
      while(temp1->next!=NULL)
      {
            temp1=temp1->next;
      }
      temp1->next=newParam;
   }

   int searchHashScope(char *sym, int x)
   {
      struct symbolTable *temp = hash[0][x];
      while(temp!=NULL)
      {
            if (strcmp(temp->symbol, sym)==0 && globalScope -1 ==
temp->scope)
            { return 0; }
            temp=temp->next;
      }
      return 1;
   }

   // This returns 0 if an identifier is already declared in the same
scope
   int checkDeclaration(char *sym)
   {
      int x = hashLocation(sym);
      if(searchHashScope(sym, x))
```

```
        return 1;
        else
        return 0;
    }

    char* getAttribute(char *sym)
    {
        char abc[20]="";
        char *ans = abc;
        int x = hashLocation(sym);
        struct symbolTable *temp = hash[0][x];
        while(temp!=NULL)
        {
              if (strcmp(temp->symbol, sym)==0 && 0 == temp->scope)
//Need to check if comparing scope with 0 is fine
              {
                    struct parameterChain *temp1 = temp->params;
                    while(temp1!=NULL)
                    {
                          //printf("\nParams are %s
%s\n",temp1->name,temp1->dataType);
                          temp1=temp1->next;
                    }
                    return temp->attribute;
              }
              temp=temp->next;
        }
        return ans;
    }

    int searchHashScope1(char *sym, int x)
    {
        struct symbolTable *temp = hash[0][x];
        while(temp!=NULL)
        {
              int i=0;
              if (strcmp(temp->symbol, sym)==0 && 0 == temp->scope)
              { return 0; }
              for (;i<=stack->top;i++)
              {
                    if (strcmp(temp->symbol, sym)==0 && stack->array[i]
== temp->scope)
                    { return 0; }
              }
              temp=temp->next;
```

```
        }
        return 1;
    }

    int checkDeclaration1(char *sym)
    {
        int x = hashLocation(sym);
        if (searchHashScope1(sym, x))
        return 1;
        else
        return 0;
    }

    void checkParameterType()
    {
        int x = hashLocation(alc2);
        struct symbolTable *temp = hash[0][x];
        if(temp==NULL)
        {
              return;
        }
        while(strcmp(temp->symbol, alc2)!=0 || temp->scope!=0)
        {
              temp=temp->next;
              if(temp==NULL)
              {
                    printf("\nTemp = null when searching for function
name in symbol table\n");
                    return;
              }
        }
        struct parameterChain *temp1;
        temp1 = temp->params;
        int i;
        for(i=0;i<parameterNumber-1;i++)
        {
              temp1=temp1->next;
              if(temp1==NULL)
              {
                    if(flag==0)
                    printf(ANSI_COLOR_RED "ERROR: Arguments passes is
more than expected\n" ANSI_COLOR_RESET);
                    return;
              }
```

```
        }

        if(flag)
        {
                if(temp1->next!=NULL)
                {
                        printf(ANSI_COLOR_RED "ERROR: Arguments passes is
less than expected\n" ANSI_COLOR_RESET);
                }
                return;
        }

        x = hashLocation(alc3);
        i=stack->top;
        struct symbolTable *temp2;
        for(;i>=0;i--)
        {
                temp2 = hash[0][x];
                while(temp2!=NULL)
                {
                if(strcmp(temp2->symbol, alc3)==0 && temp2->scope ==
stack->array[i])
                {
                        i=-1;
                        break;
                }
                temp2 = temp2->next;
                }
        }

        if(strcmp(temp1->dataType,temp2->data)!=0)
        {
                printf(ANSI_COLOR_RED "ERROR: Argument is not of the
correct type\n" ANSI_COLOR_RESET);
        }
    }

    void display()
    {
       int k=0;
       printf("\n\nSYMBOL TABLE:\n");

printf("--------------------------------------------------------------------
-----------------------------------------------------------\n");
        printf("%*s\t|\t%*s\t|\t%*s\t|\t%*s\t|\t%*s\t|\t%*s\t|\t%*s\n",
```

```
10, "INDEX", 10, "SYMBOL", 10, "ATTRIBUTE", 10, "DATATYPE", 10, "SCOPE",
10, "ProcDefFlag", 10, "NESTING");

printf("----------------------------------------------------------------
----------------------------------------------------------------\n");
	for(;k<size;k++)
	{
		struct symbolTable *temp = hash[0][k];
		while(temp!=NULL)
		{

printf("%*d\t|\t%*s\t|\t%*s\t|\t%*s\t|\t%*d\t|\t%*d\t|\t%*d\n",10,
temp->index, 10, temp->symbol, 10, temp->attribute, 10, temp->data, 10,
temp->scope, 10, temp->procedureDefinitionFlag,10,temp->nesting);
			temp = temp->next;
		}
	}

printf("----------------------------------------------------------------
----------------------------------------------------------------\n");
	k=0;
	printf("\n\nCONSTANT TABLE:\n");

printf("----------------------------------------------------------------
---\n");
	printf("%*s\t|\t%*s\t|\t%*s\n", 10, "INDEX", 10, "SYMBOL", 10,
"ATTRIBUTE");

printf("----------------------------------------------------------------
---\n");
	for(;k<size;k++)
	{
		struct symbolTable *temp = hash[1][k];
		while(temp!=NULL)
		{
			printf("%*d\t|\t%*s\t|\t%*s\n",10, temp->index, 10,
temp->symbol, 10, temp->attribute);
			temp = temp->next;
		}
	}

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

```
%%

/*The supported datatypes*/
dataType : SHORT {alc="short";}
        | INT {alc="int";}
        | LONG {alc="long";}
        | FLOAT {alc="float";}
        | DOUBLE {alc="double";}
        ;

    /*List of all statements*/
statement : declarationStatement
          | ifAndElseMatched
          | ifAndElseUnmatched
          | whileLoopStatement
          | expressionStatement
          | functionCall
          | jumpStatement
          | compoundStatement
          | ';'
          ;

statement1 : declarationStatement
          | whileLoopStatement
          | expressionStatement
          | functionCall
          | jumpStatement
          | compoundStatement
          ;

statements : statements statement
            | statement
            ;


    /*defining each type of statements*/

    /*Declaration statements: Include declaration of identifiers, with
or without initialisation*/
declarationStatement : declarationList ';'
                        | functionDeclaration
                        ;
 /* For function definition */
declaration : dataType ID { int len = strlen(yylval.sym);
```

```
                                        char *buffer=(char
*)malloc(len);
                                        int i;
                                        for (i=0;i<len-1;i++)
                                        {
                                              buffer[i] =
yyval.sym[i];
                                              alc2[i]=buffer[i];
                                        }
                                        if(checkDeclaration(buffer))
                                              {
                                                    procFlag = 1;

addToTable(0,buffer,"function", alc);
                                                    procFlag = -1;
                                              }
                                              else
                                              {

printf(ANSI_COLOR_RED "ERROR: Function is already declared\n"
ANSI_COLOR_RESET);
                                              }
                                        alc1 = alc;
                                        }
            | void ID {
                                        int len = strlen(yyval.sym);
                                        int i;
                                        for (i=0;i<len;i++)
                                        {
                                              alc2[i]=yyval.sym[i];
                                        }
                                  if(checkDeclaration(alc2))
                                  {
                                        procFlag = 1;

addToTable(0,yyval.sym,"function", alc);
                                        procFlag = -1;
                                  }
                                  else
                                  {
                                        printf(ANSI_COLOR_RED "ERROR:
Function is already declared\n" ANSI_COLOR_RESET);
                                  }
                            alc1 = alc;
                            }
```

```
            ;

 /* For argument list */
declaration1 : dataType ID {if(checkDeclaration(yylval.sym))
                                                {

addToTable(0,yylval.sym,"identifier", alc);

addParams(alc,yylval.sym);
                                                }
                                                else
                                                {

printf(ANSI_COLOR_RED "ERROR: Variable is already declared\n"
ANSI_COLOR_RESET);
                                                }
                                        };

 /* For function declaration */
declaration2 : dataType ID { int len = strlen(yylval.sym);
                                                char *buffer=(char
*)malloc(len);
                                                int i;
                                                for (i=0;i<len-1;i++)
                                                {
                                                        buffer[i] =
yylval.sym[i];
                                                        alc2[i]=buffer[i];
                                                }
                                                if(checkDeclaration(buffer))
                                                        {
                                                                procFlag = 0;

addToTable(0,buffer,"function", alc);
                                                                printf("\nadding
function to table\n");
                                                                procFlag = -1;
                                                        }
                                                        else
                                                        {

printf(ANSI_COLOR_RED "ERROR: Function is already declared\n"
ANSI_COLOR_RESET);
                                                        }
                                                }
```

```
            | void ID {
                                    int len = strlen(yylval.sym);
                                    int i;
                                    for (i=0;i<len;i++)
                                    {
                                          alc2[i]=yylval.sym[i];
                                    }
                                    if(checkDeclaration(yylval.sym))
                                    {
                                          procFlag = 0;

addToTable(0,yylval.sym,"function", alc);
                                          procFlag = -1;
                                          alc2=yylval.sym;
                                    }
                                    else
                                    {
                                          printf(ANSI_COLOR_RED "ERROR:
Function is already declared\n" ANSI_COLOR_RESET);
                                    }
                               }
            ;

declarationAndAssignment : declaration1 '=' consta
                             | declaration1 '=' stri
                             ;

completeDeclaration : declarationAndAssignment
                        | declaration1
                        ;

argumentList : argumentList ',' completeDeclaration
                  | completeDeclaration
                  ;

declarationList : dataType identi1 ',' identifierList
                  | dataType identi1
                  | dataType identi1 '=' consta
                  | dataType identi1 '=' identi
                  ;

identifierList : identifierList ',' identi1
                  | identifierList ',' identi1 '=' consta
                  | identifierList ',' identi1 '=' identi
                  | identi1
```

```
                    | identi1 '=' consta
                    | identi1 '=' identi
                    ;

/*declarationStatementError : declaration { printf("Semicolon missing
after statement %s\n",$1);}
                                | declarationAndAssignment {
printf("Semicolon missing after statement %s\n",$1);}
                                ;*/



    /*If and else statements.*/
    /*The dangling else problem is taken care by having two types of if
and else statements: Matched and unmatched statements*/
/*ifAndElseStatement : ifAndElseMatched
                       | ifAndElseUnmatched
                       ;*/
ifAndElseMatched : IF '(' expression ')' ifAndElseMatched1 ELSE
ifAndElseMatched1
                   ;
ifAndElseMatched1 : IF '(' expression ')' ifAndElseMatched1 ELSE
ifAndElseMatched1
                    | statement1
                    ;
ifAndElseUnmatched : IF '(' expression ')' ifAndElseMatched1 ELSE
ifAndElseUnmatched
                       | IF    '(' expression ')' statement1
                       ;
/*ifAndElseStatementError : IF '(' expression ifAndElseMatched ELSE
ifAndElseMatched
                            | IF expression ')' ifAndElseMatched ELSE
ifAndElseMatched
                            | IF expression ')' ifAndElseMatched ELSE
ifAndElseUnmatched
                            | IF '(' expression ifAndElseMatched ELSE
ifAndElseUnmatched
                            | IF expression ')' statement
                            | IF '(' expression statement
                            ;*/



    /*While loop*/
whileLoopStatement : WHILE whileParanthesisStart expression
```

```
whileParanthesisEnd statement
                                ;
whileParanthesisStart : '(' {whileStart=1;};
whileParanthesisEnd : ')' {whileStart=0;};
/*whileLoopStatementError : WHILE expression ')' statement
{addToTable(0,"while","keyword");}
                                | WHILE '(' expression statement
{addToTable(0,"while","keyword");}
                                ;*/


    /*Expressions are statements with operators. The expressions are
defined taking care of the precedence of operators*/
expressionStatement : expression ';'
                        ;
expression : assignment_expression
           | expression ',' assignment_expression
           ;
primary_expression
    : identi {int len = strlen(yylval.sym);
                  char *buffer=(char *)malloc(len);
                  int i;
                  for (i=0;i<len;i++)
                  {
                  if ((yylval.sym[i]>='a' && yylval.sym[i]<='z') ||
(yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                        buffer[i] = yylval.sym[i];
                  }
                  if(whileStart==1 && getDatatype(buffer)!=1)
printf(ANSI_COLOR_RED "\nERROR: Condition of while has to be int\n"
ANSI_COLOR_RESET);}
    | consta {if(rhs==0) printf(ANSI_COLOR_RED "\nERROR: Lvalue required
to be identifier\n" ANSI_COLOR_RESET);}
    | '(' expression ')'
    ;
postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' identi
    | postfix_expression INC
    | postfix_expression DEC
    ;
```

```
argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

unary_expression
    : postfix_expression
    | INC unary_expression
    | DEC unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' dataType ')'
    | SIZEOF '(' VOID ')'
    ;
unary_operator
    : '&'
    | '*'
    | '+'
    | '-'
    | '~'
    | '!'
    ;
cast_expression
    : unary_expression
    | '(' dataType ')' cast_expression
    ;
multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;
additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;
shift_expression
    : additive_expression
    | shift_expression LEFT additive_expression
    | shift_expression RIGHT additive_expression
    ;
relational_expression
    : shift_expression
```

```
    | relational_expression gt shift_expression
    | relational_expression lt shift_expression
    | relational_expression LE shift_expression
    | relational_expression GE shift_expression
    ;

gt : '>' {rhs=1;};
lt : '<' {rhs=1;};

equality_expression
    : relational_expression
    | equality_expression EQ relational_expression
    | equality_expression NE relational_expression
    ;
and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;
exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;
inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;
logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND inclusive_or_expression
    ;
logical_or_expression
    : logical_and_expression
    | logical_or_expression OR logical_and_expression
    ;
conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;
assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;
assignment_operator
    : '=' {rhs=1;}
    | MUL_ASSIGN {rhs=1;}
```

```
    | DIV_ASSIGN {rhs=1;}
    | MOD_ASSIGN {rhs=1;}
    | ADD_ASSIGN {rhs=1;}
    | SUB_ASSIGN {rhs=1;}
    | LEFT_ASSIGN {rhs=1;}
    | RIGHT_ASSIGN {rhs=1;}
    | AND_ASSIGN {rhs=1;}
    | XOR_ASSIGN {rhs=1;}
    | OR_ASSIGN {rhs=1;}
    ;


    /*Jump statements include continue,break and return statements*/
jumpStatement : CONTINUE ';'
              | BREAK ';'
              | RETURN ';' {
              if (alc1!=NULL && strcmp(alc1,"void")!=0)
printf(ANSI_COLOR_RED "\nERROR: Function type is %s return void found\n"
ANSI_COLOR_RESET, alc1);}
              | RETURN ID ';' {
                  int len = strlen(yylval.sym);
                  char *buffer=(char *)malloc(len);
                  int i;
                  for (i=0;i<len;i++)
                  {
                  if ((yylval.sym[i]>='a' && yylval.sym[i]<='z') ||
(yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                        buffer[i] = yylval.sym[i];
                  }
                  setDatatype = getDatatype(buffer);
                  if(setDatatype==1 && strcmp(alc1, "int"))
                        printf(ANSI_COLOR_RED "\nERROR: Function with
return type int returning void\n" ANSI_COLOR_RESET);
                  else if (setDatatype==0 && strcmp(alc1, "void"))
                        printf(ANSI_COLOR_RED "\nERROR: Function with
return type void returning int\n" ANSI_COLOR_RESET);
                  }
                  | RETURN consta ';' {
                  if (strcmp(alc1, "void")==0)
                        printf(ANSI_COLOR_RED "\nERROR: Function with
return type void returning value\n" ANSI_COLOR_RESET);
                  }
              | RETURN expression ';'
              ;
```

```
/*           jumpStatementError : CONTINUE
                 | BREAK
                 | RETURN
                 ;*/


    /*Compound statements are the statements enclosed within the curly
braces*/
compoundStatement : startCompound statements endCompound
                    | startCompound endCompound
                    ;

startCompound : '{' {push(stack, newScope++); globalScope = newScope;
                        push(stack1, currentScope++);
                        }
              ;
endCompound : '}' {pop(stack); globalScope =
stack->array[stack->top]+1;
                        if (stack->top == -1) globalScope = 1;
                        pop(stack1); currentScope =
stack1->array[stack1->top]+1;
                        if (stack1->top == -1) currentScope = 1;
                        }
            ;

/*compoundStatementError : '{'
                           | '}'
                           | '{' statements
                           | statements '}'
                           ;*/



startSymbol
    : external_declaration
    | startSymbol external_declaration
    ;

external_declaration
    : functionDefinition
    | declarationStatement
    | HEADERFILE
    ;

functionCall : identi3 '(' ')' ';'
```

```
            | identi3 '(' parameters ')' ';' {if(flag2==0){ flag=1;
checkParameterType(); flag=0; parameterNumber=0;}}
            ;

parameters : parameters ',' identi {parameterNumber++;
checkParameterType();}
                   | parameters ',' identi '=' stri {parameterNumber++;
checkParameterType();}
                   | parameters ',' identi '=' consta
{parameterNumber++; checkParameterType();}
                   | parameters ',' identi '=' identi
{parameterNumber++; checkParameterType();}
                   | parameters ',' consta {parameterNumber++;
checkParameterType();}
                   | parameters ',' stri {parameterNumber++;
checkParameterType();}
                   | identi { parameterNumber++; checkParameterType();}
                   | identi '=' stri {parameterNumber++;
checkParameterType();}
                   | identi '=' consta {parameterNumber++;
checkParameterType();}
                   | identi '=' identi {parameterNumber++;
checkParameterType();}
                   | consta {parameterNumber++; checkParameterType();}
                   | stri {parameterNumber++; checkParameterType();}
                   ;

functionDefinition : declaration '(' ')' compoundStatement
                        | declaration startParenthesis argumentList ')'
'{' statements endCompound {setDatatype = 0;}
                        | declaration startParenthesis argumentList ')'
'{' endCompound
                        ;

functionDeclaration : declaration2 '(' ')' ';'
                         | declaration2 startParenthesis argumentList ')'
';' {pop(stack);
                         globalScope = stack->array[stack->top]+1;
                         if (stack->top==-1)
                         {
                               globalScope = 1;
                         }
                         pop(stack1); currentScope =
stack1->array[stack1->top]+1;
```

```
                        if (stack1->top == -1) currentScope = 1;
                        }
                        ;

startParenthesis : '(' {push(stack,newScope++); globalScope = newScope;
                              push(stack1,currentScope++);}
                   ;

identi3 : ID {
                  int len = strlen(yylval.sym);
                  char *buffer=(char *)malloc(len);
                  int i;
                  for (i=0;i<len-1;i++)
                  {
                        if ((yylval.sym[i]>='a' && yylval.sym[i]<='z')
|| (yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                        {
                              buffer[i] = yylval.sym[i];
                              alc2[i]=buffer[i];
                        }
                  }
                  alc2[i]='\0';
                  if(checkDeclaration1(buffer)==0)
                  {
                        if(strcmp(getAttribute(buffer),"function")!=0)
                        {
                              printf(ANSI_COLOR_RED "\nERROR: %s is not
a function\n" ANSI_COLOR_RESET, buffer);
                        }
                        else
                        {
                        }
                  }
                  else
                  {
                        printf(ANSI_COLOR_RED "\nERROR: Function used is
not declared\n" ANSI_COLOR_RESET);
                        flag2=1;
                  }
            };

identi : ID {
                  int len = strlen(yylval.sym);
                  char *buffer=(char *)malloc(len);
```

```
                  int i;
                  for (i=0;i<len;i++)
                  {
                  if ((yylval.sym[i]>='a' && yylval.sym[i]<='z') ||
(yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                  {
                        buffer[i] = yylval.sym[i];
                        alc3[i]=buffer[i];
                  }
            }
            if(checkDeclaration1(buffer)==0)
                  {
                  }
            else
            {
                  printf(ANSI_COLOR_RED "\nERROR: Variable used is not
declared\n" ANSI_COLOR_RESET);
            }
      };

identi1 : ID {
                  int len = strlen(yylval.sym);
                  char *buffer=(char *)malloc(len);
                  int i;
                  for (i=0;i<len;i++)
                  {
                  if ((yylval.sym[i]>='a' && yylval.sym[i]<='z') ||
(yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                        buffer[i] = yylval.sym[i];
                  }

                  if(checkDeclaration(buffer))
                  {
                        addToTable(0,buffer,"identifier", alc);
                  }
                  else
                  {
                        printf(ANSI_COLOR_RED "ERROR: Variable is
already declared\n" ANSI_COLOR_RESET);
                  }
            };
```

```
consta : CONSTANT {addToTable(1,yylval.sym,"constant", "");};
stri : STR {addToTable(1,yylval.sym,"string", "");};
void : VOID {alc="void";};
%%

#include<stdio.h>
extern int lineNo;
int main()
{
    init();
    yyparse();
    if (searchMain()==0)
       printf(ANSI_COLOR_RED "ERROR: main function not present\n"
ANSI_COLOR_RESET);
    display();
}
int yywrap()
{
    return 1;
}
int yyerror(char *msg)
{
    printf("Error: %s in line %d\n",msg, lineNo);
}
```

Explanation of implementation

The lex code identifies tokens and returns them to the parser. The lex rules are written in the form of regular expressions.

The parser code takes the tokens from lexical analyser and matched them with the rules. The parser rules are written in the form of context free grammar.

The semantic code checks if the sentences formed follow the semantic rules. The semantic rules are written along with the syntax rules. They make use of global variables and symbol table columns like attribute, token type.

In ICG module three address code for the given high level C code is generated. To do that, for every instruction in C, one or more three address code instructions are generated and stored in the string. For if and while goto statements are used. The if statements are backpatched when the end of if statements are actually encountered. For function call return address and return value registers are used, which use return address of the next statement after function call and the value returning from the statement respectively.

Output and screenshots

Our program identifies various types of tokens and displays them. Initially, it prints all the tokens as it encounters them. The syntax analyser calls the scanner to fetch tokens one by one and tries to match to the rules specified in the parser. The identifiers and constants are inserted into the symbol and constant table when encountered. In the end, every token is displayed in symbol and constants table. During the syntax phase, semantic rules are also checked. This inserts a few more columns in the table. When the statements which contribute to the intermediate code is encountered, intermediate code is generated with some blanks. These blanks are filled by backpatching. We have made various test cases depicting various functionalities of our compiler.

Test case 1

This test case covers

1) If and else statement
2) Expressions

```c
#include <stdio.h>

int main()
{
    int a,b;
    if (a==2)
    {
       b=1;
    }
    else
    {
       b=2;
    }
    return 0;
}
```

Test case 2

This test case covers

1) nested If and else statements
2) Assignment during declaration

```c
#include <stdio.h>

int main()
{
    int a=5,b;
    if(a==5)
    {
       b=4;
    }
    else
    {
       if(a==5)
       {
             a =5;
       }
       else
       {
             a=10;
       }
       b=3;
    }
return 0;
}
```

```
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ ./a.out < testcases/project4/test2.c
main:
a       =       5
$t0     =       a       ==      5
if( not $t0)goto $L1
b       =       4
goto $L2
$L1:
$t1     =       a       ==      5
if( not $t1)goto $L3
a       =       5
goto $L4
$L3:
a       =       10
$L4:
b       =       3
$L2:
$ret_value = 0
goto $ret_addr
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$
```

Test case 3

This test case covers

1) Nested while loops

2) Post decrement operator
3) Pre increment operator

```c
#include <stdio.h>

int main()
{
    int a=10,b=4,c=5;
    while(a)
    {
       b = a--;
       while(b>10)
       {
              c=++b;
       }
    }
    return 0;
}
```

```
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ ./a.out < testcases/project4/test3.c
main:
a       =       10
b       =       4
c       =       5
$L1:
if( not a)goto $L2
$t0     =       a
$t1     =       a       -       1
a       =       $t1
b       =       $t0
$L3:
$t2     =       b       >       10
if( not $t2)goto $L4
$t3     =       b       +       1
b       =       $t3
c       =       $t3
goto $L3
$L4:
goto $L1
$L2:
$ret_value = 0
goto $ret_addr
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$
```

Test case 4

This test case covers

1) Function call (function returning int)

```c
int foo(int a)
{
    int x,y;
    x=y+5;
    return x;
}
int main()
{
```

```
    int a;
    a=foo(a);
    a = 5;
    foo(a);
    return 0;
}
```

Test case 5

This test case covers

1) While loop inside If statement

```
#include <stdio.h>

int main()
{
    int a=5;
    int b;
    b=6;
    if (b==3)
    {
       while(a>0)
       {
             a--;
       }
    }
    return 0;
}
```

```
x       =       $t0
$ret_value = x
goto $ret_addr
main:
$ret_addr = label1
goto foo
label1:
a       =       $ret_value
a       =       5
$ret_addr = label2
goto foo
label2:
$ret_value = 0
goto $ret_addr
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ ./a.out < testcases/project4/test5.c
main:
a       =       5
b       =       6
```

Test case 6

This test case covers

2) Array inside a while loop

```c
#include <stdio.h>

int main()
{
	int a[10];
	int i=0;
	while(i<9)
	{
	a[i]=a[i+1];
	}
}
```

```
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ ./a.out < testcases/project4/test6.c
main:
i       =       0
$L1:
$t0     =       i       <       9
if( not $t0)goto $L2
$t1     =       a       +       i
$t2     =       i       +       1
$t3     =       a       +       $t2
$t1     =       $t3
goto $L1
$L2:
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$
```