Course Code: CSPC62
Course Name: COMPILER DESIGN

Title: C Compiler for while loop with nested while loop and/or if-else constructs.

Team Members:

AMAN HARSH	:	106120010
ARPON KAPURIA	:	106120014
KUNDAN KUMAR	:	106120058
MD. ATIAB JOBAYER PURNO	:	106120068
RAHUL	:	106120092

UNDER THE GUIDANCE OF
Dr. K. SITARA
ASSISTANT PROFESSOR , NIT TRICHY.

Table of contents

INTRODUCTION

The objective of this project is to build a Mini C++ compiler using C programming language. The end product being generation of assembly level code which is previously code optimised using python programming language.

Since it is a Mini C++ compiler , it takes care of the majority of constructs in C++ language and the rest of the constructs are left for future enhancements.

It works for constructs which include loops such as **While** and constructs such as conditional statements such as **If**, **Else If**.
The various steps included in generating the optimised intermediate code includes :

- Generating symbol table after evaluating the expressions in the input file.
- Generate Abstract Syntax Tree for the code generated.
- Generate 3 address codes which are not optimised and are in the form of quadruples.
- Generate optimised code by performing basic code optimisation techniques.
- Generate Assembly language code from optimized code.

The various tools used in building this Mini C++ compiler include LEX which helps in identification of predefined patterns and helped in generating tokens accordingly. YACC was used for parsing the input for semantic meaning and generating abstract syntax trees and intermediate code generation.

ARCHITECTURE OF LANGUAGE

The following C++ constructs have been implemented :

- While Loops
- Nested While Loops
- Simple If ● If Else
- If Else ladders

The following expressions are also being handled :

- We handled all kinds of Logical Expressions which include - Logical AND , Logical OR and Logical NOT.
- We handled all kinds of Arithmetic Expressions which include - **+ , - , * , / , ++ , --**
- We handled all kinds of Boolean Expressions which include - **> , < , <= , >= , == , != .**

Along with the various expressions included , Error handling is also being included which includes :

- Error handling mechanism used for handling errors is “Panic Mode Recovery” and the synchronizing token used to identify it is “Semi colon ” i.e “ ; ” .

PHASE 1 : IMPLEMENTATION OF EACH PHASE OF COMPILER

PHASE 1A : LEXICAL ANALYSIS

- In order to implement lexical analysis phase , we make use of LEX tool which is used to generate a lexical analyser .It does this by translating set of regular expressions specified in lex.l file into a C implementation of a corresponding finite state machine i.e lex.yy.cc.
- So , we create a scanner using the LEX tool for C++ Language.This scanner transforms the source file from a stream of bytes and transforms into a series of predefined meaningful tokens.
- This scanner also takes care of removing both single and multiline comments and removes them for further analysis.
- A Global variable called “yyval” which is used to record the value of lexeme scanned and a Global variable called “yytext” is the lex variable which stores the matched string.
- The lexical analyzer built also removes any whitespaces and comments along with breaking the syntax into a series of tokens.

  If the lexical analyzer recognizes that a token is invalid then it generates an error. The invalid token recognition task is achieved when the input string does not match any rule written in the lex file.
- The rules defined in the lex file eventually are regular expressions which also have corresponding actions associated with it which are executed on a match with the input stream.
- The lexical analyser built will eventually pass these tokens to syntax analyser whenever it demands.

LEX RULES

```
#include<stdio.h>
#include<stdlib.h>
#include "y.tab.h"
#include<string.h>
int countn=0;
%}

%option yylineno
alpha [a-zA-Z]
digit [0-9]
%%

printf  {strcpy(yylval.nam.name,(yytext));return printff;}
scanf {strcpy(yylval.nam.name,(yytext));return scanff;}
int {strcpy(yylval.nam.name,(yytext));return INT;}
float {strcpy(yylval.nam.name,(yytext));return FLOAT;}
char {strcpy(yylval.nam.name,(yytext));return CHAR;}

void return VOID;
return return RETURN;
if  return IF;
else    return ELSE;
while   return WHILE;

^"#include"[ ]*<.+\.h> return INCLUDE;
"true" return TR;
"false" return FL;

{digit}+   { strcpy(yylval.nam.name,(yytext));   return NUM;}
{alpha}({alpha}|{digit})* {strcpy(yylval.nam.name,yytext); return ID;}

"<="   {strcpy(yylval.nam.name,yytext);  return LE;}
">="   {strcpy(yylval.nam.name,yytext);  return GE;}
"=="   {strcpy(yylval.nam.name,yytext); return EQ;}
```

```
"!="   {strcpy(yylval.nam.name,yytext); return NE;}
">" {strcpy(yylval.nam.name,yytext);return GT;}
"<" {strcpy(yylval.nam.name,yytext);return LT;}
"&&"    {strcpy(yylval.nam.name,yytext); return AND;}
"||"    {strcpy(yylval.nam.name,yytext);return OR;}
\/\/.* ;
\/\*(.*\n)*.*\*\/ ;
[ \t]* ;
[\n]   countn++;
. return yytext[0];
["].*["] {strcpy(yylval.nam.name,yytext);return STRLT;}

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

PHASE 1B : SYNTAX ANALYSIS

- The syntax analysis phase will analyse the syntactical structure of the input.Basically , this phase checks if the given input is in correct syntax in accordance with the programming language.
- It is responsible for verifying the sequence of tokens forms a valid sentence or not .It verifies this by making use of Programming Language Grammar defined.

The project supports these implementations :

- Variable declaration and initialization.
- Variables of type : int , float , char are supported.
- Arithmetic , Boolean and Logical expressions are supported.
- For the purpose of parsing , we use YACC which provides a tool to produce a parser for given grammar. YACC also reports shift-reduce and reduce-reduce conflicts generated on parsing an ambiguous grammar.YACC

CONTEXT FREE GRAMMAR USED

```
START P : I M ID {insert_type_table();}
   '('{add('t');}
   R
   ')'{add('t');}
   '{'{add('t');}
   S { printf("Label next:\n");}
   U
   '}'{
      $$.nd = mknode(NULL,$12.nd,"start");
      printf("\t\t\tSyntax Tree in Inorder traversal\n");
      printtree($$.nd);
      printf("\n\n");
      add('t');
      optimized();
   };
I : I I | INCLUDE {add('H');} ;

M : INT{insert_type();} | FLOAT{insert_type();}| CHAR{insert_type();} | VOID{insert_type();} ;

R : R ','{add('t');} R | M N TER | N TER;
```

```
TER : ';'{add('t');} | ;

N : ID{insert_type_table();} G | '*'{add_ptr();} N ;

G : '['{add('t');} NUM {add('n');}']' G |
   ['ID ']' G |
   '['{add('t');} ']' G |  ;

U : RETURN NUM {add('n');} ';'{add('t');printf("Return\t%s\n",$2.name);} |
   RETURN ID ';' {add('t');printf("Return\t%s\n",$2.name);}| ;

S:  S1
   |S2
   |assign {$$.nd=$1.nd;}
   |M ID TER {$$.nd=mknode(NULL,NULL,"definition"); int i=sym_search($2.name);if(i!=-1){
         if(strcmp($1.name,"int")==0){
            addTo('i',$2.name);
         }
         else if(strcmp($1.name,"float")==0)addTo('f',$2.name);
         else addTo('c',$2.name);
      }
      else{printf("Variable already defined, error at line no: %d\n",yylineno);exit(0);}
   }
   |S S {$$.nd=mknode($1.nd,$2.nd,"statement");strcpy($$.name,"STATEMENT");}
   |printff  {add('f');} '(' STRLT ')';' {$$.nd = mknode(NULL,NULL,"printf");}
   |scanff {add('f');}'('STRLT ',"&'ID')' ';'{$$.nd = mknode(NULL,NULL,"scanf");}
   |{$$.nd=mknode(NULL,NULL,"EPSILON");};

S1 :    IF{add('k');} '(' C ')'{printf("Label\t%s:\n",$4.tr);}'{'{addTo('{',"Punctuations");} S
'}'{addTo('}',"Punctuations");
   pop();
   printf("goto next");
   printf("Label\t%s:\n",$4.fal);} EL {$$.nd=mknode($4.nd,$9.nd,"IF");
   strcpy($$.name,"IF");}
   |assign {$$.nd=$1.nd;}
   |M ID TER {$$.nd=mknode(NULL,NULL,"definition"); int i=sym_search($2.name);if(i!=-1)
   {if(strcmp($1.name,"int")==0){addTo('i',$2.name);}
```

```
    else if(strcmp($1.name,"float")==0)addTo('f',$2.name);
    else addTo('c',$2.name);}
    else{printf("Variable already defined, error at line no: %d\n",yylineno);exit(0);}}
    |S S {$$.nd=mknode($1.nd,$2.nd,"statement");strcpy($$.name,"STATEMENT");}
    |printff  {add('f');} '(' STRLT ')'';' {$$.nd = mknode(NULL,NULL,"printf");}
    |scanff {add('f');}'('STRLT ','&'ID')' ';'{$$.nd = mknode(NULL,NULL,"scanf");}
    |{$$.nd=mknode(NULL,NULL,"EPSILON");};

S2:
    WHILE{add('k'); sprintf(nL,"L%d",ifd);ifd++;printf("\n Label \t %s : \n",nL);} '(' C ')'{printf("\n
Label\t%s:\n",$4.tr);}'{'{addTo('{',"Punctuations");} S '}'{addTo('}',"Punctuations");
    pop();
    printf("goto %s\n",nL);
    printf("Label\t%s:\n",$4.fal);} EL {$$.nd=mknode($4.nd,$9.nd,"WHILE");
    strcpy($$.name,"WHILE");}
    |assign {$$.nd=$1.nd;}
    |M ID TER {$$.nd=mknode(NULL,NULL,"definition"); int i=sym_search($2.name);if(i!=-1)
    {if(strcmp($1.name,"int")==0){addTo('i',$2.name);}
    else if(strcmp($1.name,"float")==0)addTo('f',$2.name);
    else addTo('c',$2.name);}
    else{printf("Variable already defined, error at line no: %d\n",yylineno);exit(0);}}
    |S S {$$.nd=mknode($1.nd,$2.nd,"statement");strcpy($$.name,"STATEMENT");}
    |printff  {add('f');} '(' STRLT ')'';' {$$.nd = mknode(NULL,NULL,"printf");}
    |scanff {add('f');}'('STRLT ','&'ID')' ';'{$$.nd = mknode(NULL,NULL,"scanf");}
    |{$$.nd=mknode(NULL,NULL,"EPSILON");};

EL: ELSE{add('k');} '{'{addTo('{',"Punctuations");} S '}' {$$=$5;addTo('}',"Punctuations");
    pop();
    printf("goto next\n");
    printf("\n");}| {printf("goto next\n");
    printf("\n");};

Arg : STRLT ;

C : C AND B | C OR B | NE B | B{$$.nd=$1.nd;};

B :     E relop E {$$.nd=mknode($1.nd,$3.nd,$2.name);
    int i=search($1.name);
    int j=search($3.name);
```

```
 if(i!=0&&j!=0){
 printf("if %s %s %s goto L%d \nelse goto L%d\n",$1.name,$2.name,$3.name,ifd,eld);
sprintf($$.tr,"L%d",ifd);
sprintf($$.fal,"L%d",eld);ifd++;eld++;}
else{printf(" Variable not declared at line no: %d\n", yylineno);exit(0);}}
| ID '=' {add('o');} E{int i=search($1.name);
 int j=search($4.name);
 if(i!=0&&j!=0)
 {
 printf("if %s!=0 goto L%d else goto L%d\n",$1.name,ifd,eld);
 sprintf($$.tr,"L%d",ifd);
 sprintf($$.fal,"L%d",eld);ifd++;eld++;}
 else{printf(" Variable not declared at line no: %d\n", yylineno);
 exit(0);}}
 | FL{printf("if False  goto L%d\n",eld);
 sprintf($$.tr,"L%d",ifd);
 sprintf($$.fal,"L%d",eld);ifd++;eld++;}
 | TR {printf("if True  goto L%d\n",ifd);
 sprintf($$.tr,"L%d",ifd);
 sprintf($$.fal,"L%d",eld);ifd++;eld++;}
 | ID {int i=search($1.name);
 if(i!=0)
 {
 printf("if %s!=0  goto L%d else goto L%d\n",$1.name,ifd,eld);
 sprintf($$.tr,"L%d",ifd);
 sprintf($$.fal,"L%d",eld);ifd++;eld++;}
 else {printf(" Variable not declared at line no: %d\n", yylineno);
 exit(0);} }
 | NUM {add('n');
 printf("if %s!=0  goto L%d else goto L%d\n",$1.name,ifd,eld);
 sprintf($$.tr,"L%d",ifd);
 sprintf($$.fal,"L%d",eld);ifd++;eld++;} ;

assign :  ID '='{add('o');} E ';'{$1.nd = mknode(NULL,NULL,$1.name);
    $$.nd=mknode($1.nd,$4.nd,"=");
    strcpy($$.name,"=");add('t');
```

```
    int i=search($1.name);
    int j=search($4.name);
    if(i!=0&&j!=0)
    {
    type_check($1.name,$4.name);
    printf("= \t %s\t %s \n",$4.name,$1.name);
      strcpy(intermediate_code[code].op,"=");
      strcpy(intermediate_code[code].res,$1.name);
      strcpy(intermediate_code[code].op1,$4.name);
      code++;
      }
    else {printf("Variable not declared at line no: %d\n", yylineno);
    exit(0);}} |ID '(' Arg ')' ';'{add('t');};

E : E '+'{add('o');} E {$$.nd=mknode($1.nd,$4.nd,"+");strcpy($$.name,"+");
   int i=search($1.name);
   int j=search($4.name);
   sprintf($$.name,"t%d",c);c++;
   addTo(temptype($1.name,$4.name),$$.name);
   if(i!=0 && j!=0)
{printf("%s\t%s\t%s\t%s\n","+",$1.name,$4.name,$$.name);strcpy(intermediate_code[code].op,"+");
      strcpy(intermediate_code[code].res,$$.name);
      strcpy(intermediate_code[code].op1,$1.name);
      strcpy(intermediate_code[code].op2,$4.name);
      code++;}
    else {printf(" Variable not declared at line no: %d\n", yylineno);exit(0);}}

   | E '-' {add('o');}E { $$.nd=mknode($1.nd,$4.nd,"-");
   strcpy($$.name,"-");
   int i=search($1.name);
   int j=search($4.name);
   sprintf($$.name,"t%d",c);c++;
   addTo(temptype($1.name,$4.name),$$.name);
   if(i!=0 && j!=0) {printf("%s\t%s\t%s\t%s\n","-",$1.name,$4.name,$$.name);
strcpy(intermediate_code[code].op,"-");
      strcpy(intermediate_code[code].res,$$.name);
      strcpy(intermediate_code[code].op1,$1.name);
      strcpy(intermediate_code[code].op2,$4.name);
      code++;}
```

```
   else {printf(" Variable not declared at line no: %d\n", yylineno);exit(0);}}
   | F{$$.nd=$1.nd;};

F :  F '*'{add('o');} F {
   $$.nd=mknode($1.nd,$4.nd,"*");
   strcpy($$.name,"*");
   int i=search($1.name);
   int j=search($4.name);
   sprintf($$.name,"t%d",c);c++;
   addTo(temptype($1.name,$4.name),$$.name);
   if(i!=0 && j!=0) {
      printf("%s\t%s\t%s\t%s\n","*",$1.name,$4.name,$$.name);strcpy(intermediate_code[code].op,"*");
      strcpy(intermediate_code[code].res,$$.name);
      strcpy(intermediate_code[code].op1,$1.name);
      strcpy(intermediate_code[code].op2,$4.name);
      code++;}
   else{
      printf(" Variable not declared at line no: %d\n", yylineno);exit(0);
   }
}

   | F '/' {add('o');}F {$$.nd=mknode($1.nd,$4.nd,"/");strcpy($$.name,"/");
   int i=search($1.name);
   int j=search($4.name);
   sprintf($$.name,"%d",c);
   strcat($$.name,"t");c++;
   addTo(temptype($1.name,$4.name),$$.name);
   if(i!=0 && j!=0) {
      printf("%s\t%s\t%s\t%s\n","/",$1.name,$4.name,$$.name);strcpy(intermediate_code[code].op,"/");
      strcpy(intermediate_code[code].res,$$.name);
      strcpy(intermediate_code[code].op1,$1.name);
      strcpy(intermediate_code[code].op2,$4.name);
      code++;
   }
    else {printf(" Variable not declared at line no: %d\n", yylineno);
    exit(0);}}
```

```
   | T {$$.nd=$1.nd;};

T : T '^'{add('o');} T {$$.nd=mknode($1.nd,$4.nd,"^");
   strcpy($$.name,"^");
   int i=search($1.name);
   int j=search($4.name);
   sprintf($$.name,"%d",c);
   strcat($$.name,"t");c++;
   addTo(temptype($1.name,$4.name),$$.name);
   if(i!=0 && j!=0) {
   printf("%s\t%s\t%s\t%s\n","^",$1.name,$4.name,$$.name);strcpy(intermediate_code[code].op,"^");
      strcpy(intermediate_code[code].res,$$.name);
      strcpy(intermediate_code[code].op1,$1.name);
      strcpy(intermediate_code[code].op2,$4.name);
      code++;}
   else {printf(" Variable not declared at line no: %d\n", yylineno);exit(0);}}

   | Q {$$.nd=$1.nd;};

Q : '('{add('t');} E ')'{add('t'); $$=$3;} | ID {insert_type_table();} G
{$$.nd=mknode(NULL,NULL,$1.name);strcpy($$.name,$1.name);}|
NUM{add('n');}{$$.nd=mknode(NULL,NULL,$1.name);
   strcpy($$.name,$1.name);} ;

relop : LE {add('r');} | GE {add('r');}| LT {add('r');} | GT {add('r');}| EQ {add('r');};

%%
```

PHASE 2 : GENERATION OF SYMBOL TABLE ALONG WITH EXPRESSION EVALUATION

- For the generation of symbol tables , we maintain a structure which keeps track of all necessary objects which is further required for analysis.
- The structure of the symbol table contains variable name , Line number , type , value , scope. The structure is as follows :

Structure: struct
var
{ char var_name[20]; char Line_t[100]; // char type[100]; int scope;
}; struct symbol_table_entry
{ struct var arr[20]; int up;
};

- As each line is parsed , the actions associated with grammar rules are executed.
- " $1 " is used to represent the first token in the production and " $$ " is used to represent the resultant of the given production.
- When the parsing is done , a symbol table will be data-structure will be generated which we won't show on console, but a snapshot is given of how it looks like.

PHASE 3 : GENERATION OF ABSTRACT SYNTAX TREE

- For the generation of abstract syntax trees , a tree structure which represents the syntactical flow of the code is created.
- For expressions , associativity is indicated using %left and %right fields.
- For the case of precedence of operations , the last rule gets higher precedence.
- To build a tree , a structure is maintained which has 2 pointers which points to its children and token field for storage.

```
typedef struct ASTNode
{ struct ASTNode *left; struct
        ASTNode *right; char
        *token;
} node;
```

- When every new token is encountered during parsing , the function takes the value of the token and then creates a node for the tree and then makes sure that it attaches to its parent.

SNAPSHOT OF ABSTRACT SYNTAX TREE GENERATED

```
################################################################################
                    Syntax Tree in Inorder traversal
################################################################################
       start ,        definition ,    statement ,     definition ,    statement ,     definition
 ,    statement ,     definition ,    statement ,     definition ,    statement ,     definition ,
    statement ,     a ,    < ,      10 ,    WHILE ,        a ,     = ,     a ,     + ,     1 ,
  statement ,     b ,     < ,     10 ,   IF ,     c ,     = ,     c ,     + ,     1 ,     statemen
t ,     d ,     < ,     10 ,    WHILE ,        d ,    = ,     d ,     + ,     1 ,     statement
,     e ,     < ,     10 ,    IF ,    f ,     = ,     f ,     + ,    1 ,
```

PHASE 4 : INTERMEDIATE CODE GENERATION

- Intermediate code generation phase takes its input from semantic analyzer phase in the form of annotated syntax tree.This syntax tree is then converted into linear representation.
- Three Address Code : A statement which involves no more than 3 references (two for operands and one for result).A sequence of three address statements is known as three address code.
- It is of the form : a = b op c where a,b,c will have memory location.
  Example : The three address code for the expression : x * y + z will be :

```
T1 = x * y
T2 = T1 + z
T3 = T2
```

  Where T1,T2,T3 are temporary variables.
- The data structure used to represent three address codes is Quadruples which is a structure in C programming language. It has 4 columns namely : operator , operand1 ,operand2 and result.
- Due to the usage of Quadruples , our code may contain a lot of temporary variables which can increase time and space complexity.
- On the other hand , usage of Quadruples makes it easy to rearrange code for global optimization.

SNAPSHOTS OF INTERMEDIATE CODE GENERATION

```
 Label   L0 :
if a < 10 goto L1
else goto L20

 Label  L1:
+       a       1       t0
=        t0      a
if b < 10 goto L2
else goto L21
Label   L2:
+       c       1       t1
=        t1      c
goto nextLabel  L21:
+       b       1       t2
=        t2      b
goto next

goto L0
Label   L20:
goto next


 Label   L3 :
if d < 10 goto L4
else goto L22

 Label  L4:
+       d       1       t3
=        t3      d
if e < 10 goto L5
else goto L23
Label   L5:
+       f       1       t4
=        t4      f
goto nextLabel  L23:
+       e       1       t5
=        t5      e
goto next

goto L3
Label   L22:
goto next
```

PHASE 5 : CODE OPTIMIZATION

- The optimization method used is constant folding , common subexpression elimination followed by dead code elimination. This uses the fact that the compiler can implicitly do some computations.
- Constant Folding : Expressions having constant operands can be evaluated at run time and thereby increasing the performance of the code.
- Common subexpression elimination: Here , we substitute the values of known variables in the expressions which enables the code to assign static values which is better and faster than looking up and copying values of variables in the register and hence increasing the performance of the code.
- By performing constant folding and constant propagation , dead code elimination is also taken care of to an extent.

SNAPSHOT OF CODE AFTER OPTIMISATION

```
        t3 = a - 6
        t3 = b
        t4 = a + 1
        a = t4
        t7 = c
        a = t7
        t8 = b
        a = t8
        t9 = -c
        b = t9
        t13 = True
        d = t13
        t15 = g + 1
        g = t15
        t16 = t3
        g = t16


Optimization done by eliminating 12 lines.
```

PHASE 6 : ASSEMBLY CODE GENERATION

- The final phase of the compiler design is assembly code generation which takes input as intermediate code generated along with the symbol table and then outputs target assembly code.
- This phase proceeds on the assumption that the input it gets is free of any kind of errors including syntax and semantic errors.
- Allowed operations to be used for code generation include **MOV , LD , ST , ADD , SUB , CMP , BLT , BGE , BLE , BGT ,BEQ , BNE** and many more.
- Some of the design issues we face during code generation which includes Register allocation strategy , Choice of evaluation order and Instruction selection.
- For the sake of our project , the number of registers available for disposal is 16.
- While converting code from intermediate code generation to assembly code , it is also necessary to preserve the semantics of the source program.
- Assembly code generated can directly be converted to binary form i.e 0's and 1's by making use of opcode.
- During this phase , we also need to make sure that we optimally use the available registers since we only have limited registers available. The techniques employed for optimally using available registers :
- If registers are not going to be used again in the current block , then we clear the register.
- If we encounter block shifts , then we release registers.

SNAPSHOT OF GENERATED ASSEMBLY CODE

```
========================================
Machine Code generated below:
========================================
Start:
LD R1 a
ADD R2 R1 #1
MOV R1 R2
ST t0 R2
ST a R1
LD R1 c
ADD R2 R1 #1
MOV R1 R2
ST t1 R2
ST c R1
LD R1 b
ADD R2 R1 #1
MOV R1 R2
ST t2 R2
ST b R1
LD R1 d
ADD R2 R1 #1
MOV R1 R2
ST t3 R2
ST d R1
LD R1 f
ADD R2 R1 #1
MOV R1 R2
ST t4 R2
ST f R1
LD R1 e
ADD R2 R1 #1
MOV R1 R2
ST t5 R2
ST e R1
```

SAMPLE INPUTS & OUTPUTS

INPUT1

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

int main(int argv,char *argc[])
{
int a;
int b;
int c;
int d;
int e;
int f;
int i;
int j;
while(a<10)
{
    i=a+2;
    j=b+2;
    a=a+1;
    if(b<10)
    {
            while(d<10)
            {
                    i=a+2;
                    j=b+2;
                    d=d+1;
                    if(e<10)
                    {
                            f=f+1;
                    }
                    else
                    {
                            e=e+1;
                    }
            }
    }
```

```
    else
    {
            b=b+1;
    }
}
}
```

OUTPUT1:

```
#################################################################################
#########
                Intermediate code
#################################################################################
#########

 Label   L0 :
if a < 10 goto L1
else goto L20

 Label  L1:
+      a      2      t0
=       t0     i
+      b      2      t1
=       t1     j
+      a      1      t2
=       t2     a
if b < 10 goto L2
else goto L21
Label   L2:

 Label   L3 :
if d < 10 goto L4
else goto L22

 Label  L4:
+      a      2      t3
=       t3     i
```

```
+       b       2       t4
=       t4      j
+       d       1       t5
=       t5      d
if e < 10 goto L5
else goto L23
Label   L5:
+       f       1       t6
=       t6      f
goto nextLabel  L23:
+       e       1       t7
=       t7      e
goto next

goto L3
Label   L22:
goto next

goto nextLabel  L21:
+       b       1       t8
=       t8      b
goto next

goto L3
Label   L20:
goto next

Label next:
```

##

Syntax Tree in Inorder traversal

##

start , definition , statement , definition , statement , definition , statement , definition , statement , definition , statement , definition , statement , definition , statement , a , < , 10 , WHILE , i , = , a , + , 2 , statement , j , = , b , + , 2 , statement , a , = , a , + , 1 , statement ,

b , < , 10 , IF , d , < , 10 , WHILE , i , = , a , + , 2 , statement , j , = , b , + , 2 , statement , d , = , d , + , 1 , statement , e , < , 10 , IF , f , = , f , + , 1 ,

##

Quadruples

##

Operator	Operand1	Operand2	Result
+	a	2	t0
=	t0	(null)	i
+	b	2	t1
=	t1	(null)	j
+	a	1	t2
=	t2	(null)	a
+	a	2	t3
=	t3	(null)	i
=	j	(null)	t4
=	t4	(null)	j
+	d	1	t5
=	t5	(null)	d
+	f	1	t6
=	t6	(null)	f
+	e	1	t7
=	t7	(null)	e
+	b	1	t8
=	t8	(null)	b

##

Parsing is Successful

```
################################################################################
#########
                    Parser
################################################################################
#########

            symbol      type        identify        line number
--------------------------------------------------------------------------------
_________
        #include<stdio.h>              Header          0
        #include<stdlib.h>             Header          1
        int             N/A         KEYWORD         3
        main            int         IDENTIFIER      3
        (               N/A         Punctuation     3
        argv            int         IDENTIFIER      3
        ,               N/A         Punctuation     3
        char            N/A         KEYWORD         3
        argc            char*       IDENTIFIER      3
        ]               N/A         Punctuation     3
        )               N/A         Punctuation     3
        {               N/A         Punctuation     4
        ;               N/A         Punctuation     5
        a               int         variable        5
        b               int         variable        6
        c               int         variable        7
        d               int         variable        8
        e               int         variable        9
        f               int         variable        10
        i               int         variable        11
        j               int         variable        12
        while           N/A         KEYWORD         13
        <               int         IDENTIFIER      13
        10              int         NUMBER          13
        }               N/A         Punctuation     40
```

Generated ICG given as input for optimization:

```
t0 = a
t1 = b
```

t2 = c
t3 = d
t4 = e
t5 = f
t6 = i
t7 = j
t8 = a < 10
t9 = a + 2
i = t9
t10 = b + 2
j = t10
t11 = a + 1
a = t11
t12 = b < 10
t13 = d < 10
t14 = a + 2
i = t14
t15 = b + 2
j = t15
t16 = d + 1
d = t16
t17 = e < 10
t18 = f + 1
f = t18
t19 = e + 1
e = t19
t20 = b + 1
b = t20

ICG after eliminating common subexpressions:

t0 = a
t1 = b
t2 = c
t3 = d

t4 = e
t5 = f
t6 = i
t7 = j
t8 = a < 10
t9 = a + 2
i = t9
t10 = b + 2
j = t10
t11 = a + 1
a = t11
t12 = b < 10
t13 = d < 10
t14 = t9
i = t14
t15 = t10
j = t15
t16 = d + 1
d = t16
t17 = e < 10
t18 = f + 1
f = t18
t19 = e + 1
e = t19
t20 = b + 1
b = t20

ICG after constant folding:

t0 = a
t1 = b
t2 = c
t3 = d
t4 = e
t5 = f
t6 = i
t7 = j
t8 = a < 10

```
t9 = a + 2
i = t9
t10 = b + 2
j = t10
t11 = a + 1
a = t11
t12 = b < 10
t13 = d < 10
t14 = t9
i = t14
t15 = t10
j = t15
t16 = d + 1
d = t16
t17 = e < 10
t18 = f + 1
f = t18
t19 = e + 1
e = t19
t20 = b + 1
b = t20
```

Optimized ICG after dead code elimination:

```
t9 = a + 2
i = t9
t10 = b + 2
j = t10
t11 = a + 1
a = t11
t14 = t9
i = t14
t15 = t10
j = t15
t16 = d + 1
```

```
d = t16
t18 = f + 1
f = t18
t19 = e + 1
e = t19
t20 = b + 1
b = t20
```

Optimization done by eliminating 12 lines.

==========

Machine Code generated below:

==========

```
Start:
LD R1 a
ADD R2 R1 #2
MOV R3 R2
ST t0 R2
LD R2 b
ADD R4 R2 #2
MOV R5 R4
ST t1 R4
ADD R4 R1 #1
MOV R1 R4
ST t2 R4
ADD R4 R1 #2
ST a R1
MOV R3 R4
ST t3 R4
ST i R3
MOV R1 R5
MOV R5 R1
ST t4 R1
ST j R5
LD R1 d
ADD R3 R1 #1
MOV R1 R3
ST t5 R3
```

```
ST d R1
LD R1 f
ADD R3 R1 #1
MOV R1 R3
ST t6 R3
ST f R1
LD R1 e
ADD R3 R1 #1
MOV R1 R3
ST t7 R3
ST e R1
ADD R1 R2 #1
MOV R2 R1
ST t8 R1
ST b R2
```

INPUT2

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

int main(int argv,char *argc[])
{
int a;
int b;
int c;
int d;
int e;
int f;
int i;
int j;
while(a<10)
{
    i=a+2;
    j=b+2;
    a=a+1;
```

```
    if(b<10)
    {
            while(d<10)
            {
                    i=a+2;
                    j=b+2;
                    d=d+1;
                    else
                    {
                            e=e+1;
                    }
            }
    }
    else
    {
            b=b+1;
    }
}
}
```

OUTPUT2:

```
##################################################################################
#########
                Intermediate code
##################################################################################
#########

 Label   L0 :
if a < 10 goto L1
else goto L20

 Label  L1:
+       a       2       t0
=       t0      i
+       b       2       t1
=       t1      j
+       a       1       t2
=       t2      a
```

```
if b < 10 goto L2
else goto L21
Label   L2:

 Label   L3 :
if d < 10 goto L4
else goto L22

 Label  L4:
+       a       2       t3
=        t3      i
+       b       2       t4
=        t4      j
+       d       1       t5
=        t5      d
```

Not accepted

Reason: According to C language there should be a if block before defining an else block. In our input we didn't provide an if block. So, while parsing when the pointer reached else block inside while loop it didn't find if block before else so it called yyerror() function and which printed *"NOT ACCEPTED"* and exited the program.

INPUT3

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

int main(int argv,char *argc[])
{
int a;
int b;
int c;
int d;
int e;
```

```
int f;
int i;
int j;
(a<10)
{
    i=a+2;
    j=b+2;
    a=a+1;
    if(b<10)
    {
            while(d<10)
            {
                    i=a+2;
                    j=b+2;
                    d=d+1;
                    else
                    {
                            e=e+1;
                    }
            }
    }
    else
    {
            b=b+1;
    }
}
}
```

OUTPUT3:

Intermediate code

Label next:

Not accepted

CONCLUSION

Through this compiler we can parse any while with a nested while and if-else constructs. Though it can be improved by adding more features to error reporting. For now, this compiler can detect error and handle it. But we can add error reporting feature like where the error was reported, which line etc. It is specifically made for C/C++ language.

GitHub Link

https://github.com/aaaaarrp/C-Compiler-while-nested-while-if-else

To run the compiler, instructions are provided in README.md file.