COMPILER DESIGN PROJECT REPORT

MINOR PROJECT

INTERMEDIATE CODE GENERATION FOR IF ELSE CONSTRUCT

SUBMITTED BY

Sangamesh (1MS09CS085)

Avinash S (1MS09CS126)

Nishanth T (1MS09CS131)

Under the Guidance of

Prof. S M Narayana

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING,

M.S.RAMAIAH INSTITUTE OF TECHNOLOGY, BANGALORE

OBJECTIVE:

The main objective of the project is to generate the intermediate code for the given IF-ELSE Statement. It consists of four main sections , the conditional section inside the IF statement , the THEN part of the statement which the code enters if the given IF condition is true , the ELSE part of the statement which the code enters if the given IF condition is false , and the statements after the ELSE construct which is evaluated after either the IF or ELSE part.

The program is modularized into three separate levels , the condition part , where in it moves to different levels of code within the condition part as per the different Boolean operators . The inside of the IF and ELSE statements move into different intermediate codes as per different assignment statements.

The programs are written in lex and yacc program.
Lex helps you by taking a set of description of possible tokens and producing a C routine, which we call as lexical analyzer or a lexer, that identify the tokens. As input is divided into tokens, a program often needs to establish the relationship among the tokens. A compiler needs to find the expression, statement, declaration, blocks and procedure in the program. This task is known as parsing and the list of rules that define the relationship that the program is a grammar. Yacc takes a concise description of a grammar and produces a C routine that can parse the grammar, a parser. The yacc parser automatically detects whenever a sequence of input token matches one of the rule of the grammar and also detects a syntax error whenever its input does not match any of the rule. When a task involves dividing the input into units and establishing some relationship among those units then we should use lex and yacc.

Source Code:

Lex Program:

DESCIPTION:

Lex is a tool for building lexical analyzer or lexer. A lexer takes an arbitrary input stream and tokenizes it. In this program, when we create a lex specification, we create a set of pattern which lex matches against the input. Lex program basically has three sections one is definition section another is rule section and last one is user subroutine section which in this case is not written here but in yacc program. In the definition section we have defined the header files and also y.tab.h which helps it to link up with the yacc program. The rules section contain the patterns that has to be matched and followed by an action part, which returns the tokens to the yacc program when the input is parsed and the pattern is matched.

PROGRAM:

```
%{
#include"y.tab.h"
%}
ALPHA [A-Za-z] // variable defined for alphabets

DIGIT [0-9] // variable defined for integer numbers
%%
if          return IF; /* rule section to return the tokens scanned */
then                 return THEN;
else                 return ELSE;
{ALPHA}({ALPHA}|{DIGIT})return ID;// regular expression for identifiers
{DIGIT}+             {yylval=atoi(yytext); return NUM;}
                                   ;}//regular expression for digits
[ \t]                 ;
\n                yyterminate();
.                 return yytext[0];
%%
```

Yacc program:

DESCRIPTION:

Yacc takes a grammar that we specify and writes a parser that recognizes valid syntax in that grammar. Grammar is a series of rules that the parser uses to recognize syntactically valid input. Again in same way as lex specification, a yacc grammar has three part structure. The first section is the definition section, handles control information for the yacc generated by the parser. The second section contains the rule of parser and the third section contains the C code. In the following program, the definition section has the header files that will generate the necessary files needed to run the C program next comes the declaration of the tokens which are passed from the lex program. Then we define the grammar for the valid input typed by the user. Then it contains the necessary SDT's (Syntax Directed Translations) which move to a specific function when that particular input is encountered by the parser . The conditional code moves as per the Boolean expression defined into various levels . The number of labels generated is equal to the number of the Boolean comparisons in the conditional code . In the IF and THEN part of the statement , an intermediate code for the corresponding assignment statements is generated . After which it jumps to the ending of the code.

PROGRAM:

```
%token ID NUM IF THEN ELSE
%right '='
%left '+' '-' ; priority from left to right
%left '*' '/'
%left UMINUS
%%

S : IF '(' Y ')'{lab();} THEN '{' X '}'{lab2();} ELSE '{' X '}'
{lab3();}   //SDT for accepting the IF-THEN-ELSE construct
  ;
X : E ';'|X X;
Y : B {abc();codegen_assigna();first();}// SDT for just one condition
  | B '&''&'{abc();codegen_assigna();second();} Y ; SDT for '&&'
                        //for boolean expressions(relative sub-expressions)
  | B {abc();codegen_assigna();third();}'|''|' Y // SDT for '||'
  | '!'B{abcde();codegen_assigna();first();} // SDT for '!'
  ;

B : V '='{push();}'='{push();}D // SDT for comparison operator
  | V '>'{push();}F ; SDT for '>' logical operator
  | V '<'{push();}F
  | V '!'{push();}'='{push();}D
  |'(' B ')'
  | V{pushab();}

;
F :'='{push();}D
  |D{pusha();}
;
D :NUM{push();} // left side of expression is a number or a identifier
  |ID{push();}
;
E :V '='{push();} E{codegen_assign();}//SDT for assignment operator
  | E '+'{push();} E{codegen();}
  | E '-'{push();} E{codegen();}
  | E '*'{push();} E{codegen();}
  | E '/'{push();} E{codegen();}
  | '(' E ')'
  | '-'{push();} E{codegen_umin();} %prec UMINUS
  | V
  | NUM{push();}
  | S
;
V : ID {push();}
  ;
%%

#include "lex.yy.c" // header files
#include<ctype.h>
char st[100][10];
int top=0;
char i_[2]="0";
char temp[2]="t";
int abcd=0;
int label[20];
int lnum=0;
int ltop=0;
int i=0;
main()
 {
 printf("Enter the expression : ");
 yyparse();
 }

pusha() // an array of characters is used as a stack
{
strcpy(st[++top]," " );
}
pushab()
{
strcpy(st[++top]," ");
strcpy(st[++top]," ");
strcpy(st[++top]," ");
}
push()
 {
  strcpy(st[++top],yytext);
 }

abc()
{
abcd++;
printf("\nX%d : if ",abcd);
}
abcde()
{
abcd++;
printf("\nX%d :not ",abcd);
}
second() // used for Boolean AND condition
{
int xyz=0;
xyz=abcd+1;
printf("falg=true else flag=false");
printf("\n if flag(true) goto x%d",xyz);
printf("\n if flag(false) goto L1");
}
first() // used for single Boolean condition
{
printf("flag=true else flag=false");
printf("\n if flag(true) goto  L0");
printf("\n if flag(false) goto L1");
}
third() // used for Boolean OR condition
{
int xyz=0;
xyz=abcd+1;
printf("flag=true else flag=false");
printf("\n if flag(true) goto L0 ");
printf("\n if flag(false) goto x%d",xyz);
}
codegen() //used to print out the pushed elements in the stack
 {
 strcpy(temp,"t");
 strcat(temp,i_);
  printf("%s = %s %s %s\n",temp,st[top-2],st[top-1],st[top]);
  top-=2;
 strcpy(st[top],temp);
 i_[0]++;
 }

codegen_umin()
 {
 strcpy(temp,"t");
 strcat(temp,i_);
 printf("%s = -%s\n",temp,st[top]);
 top--;
 strcpy(st[top],temp);
 i_[0]++;
 }

codegen_assigna() // used to pop from the stack used by user
{
printf("%s %s %s %s ",st[top-3],st[top-2],st[top-1],st[top]);
top-=3;
}

lab()//semantic action after IF is encountered in the input statement
{
printf("\nL0 :\n");
}

lab2() // semantic action for THEN part of IF statement
{
int x;
lnum++;
x=label[ltop--];
printf("goto L2\n");
printf("L%d: \n",++x);
label[++ltop]=lnum;
}

lab3() // semantic action for ELSE part of IF statement
{
int y;
y=label[ltop--];
printf("L2: \n");
}
```

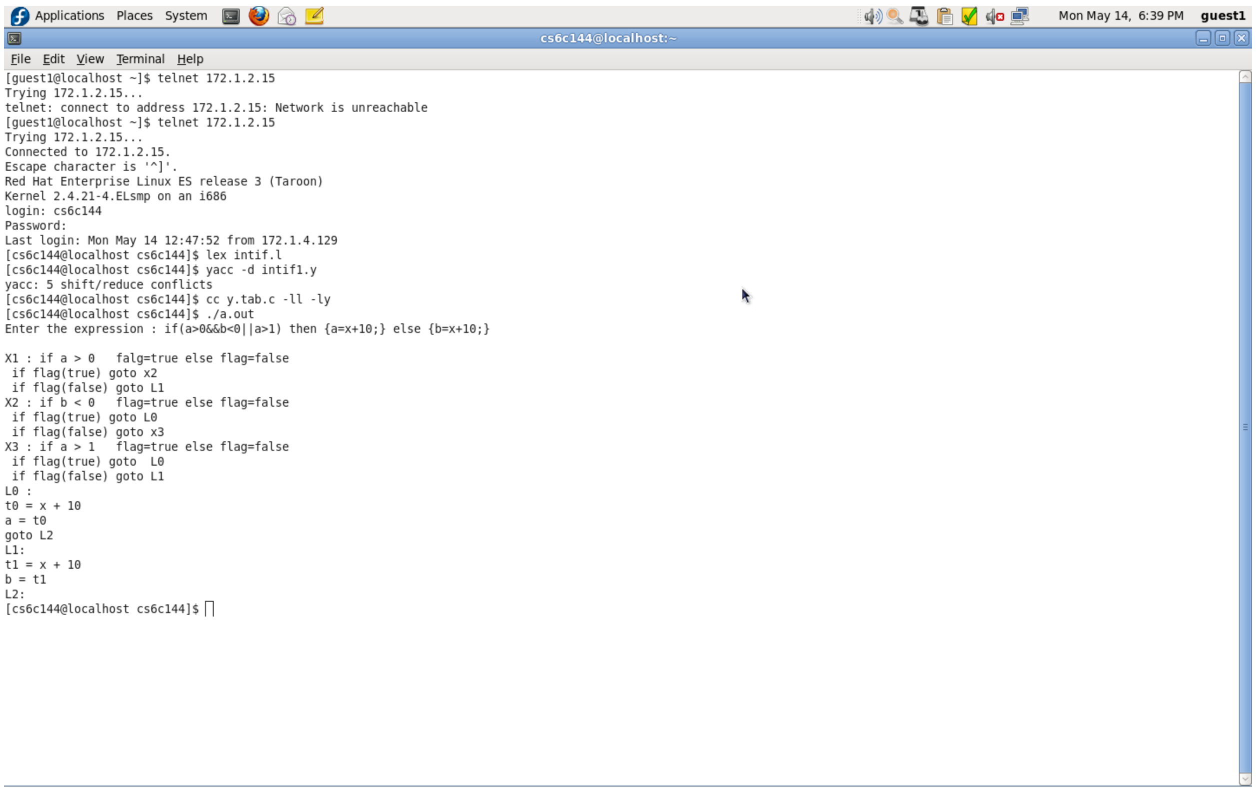
INPUT:

```
If(a>0&&b<0||a<1) then {a=x+10;} else {b=x+10;}
```

OUTPUT:

```
X1: if a > 0 flag=true else flag=false
  If flag(true) goto X2
  If flag(false) goto L1
X2: if b < 0 flag=true else flag=false
  If flag(true) goto L0
  If flag(false) goto X3
X3: if a < 1 flag=true else flag=false
  If flag(true) goto L0
  If flag(false) goto L1
L0:
T0=x + 10
a=t0
L1:
t1 =x + 10
b = t1
L2:
```

Snapshot:



```
[guest1@localhost ~]$ telnet 172.1.2.15
Trying 172.1.2.15...
telnet: connect to address 172.1.2.15: Network is unreachable
[guest1@localhost ~]$ telnet 172.1.2.15
Trying 172.1.2.15...
Connected to 172.1.2.15.
Escape character is '^]'.
Red Hat Enterprise Linux ES release 3 (Taroon)
Kernel 2.4.21-4.ELsmp on an i686
login: cs6c144
Password:
Last login: Mon May 14 12:47:52 from 172.1.4.129
[cs6c144@localhost cs6c144]$ lex intif.l
[cs6c144@localhost cs6c144]$ yacc -d intif1.y
yacc: 5 shift/reduce conflicts
[cs6c144@localhost cs6c144]$ cc y.tab.c -ll -ly
[cs6c144@localhost cs6c144]$ ./a.out
Enter the expression : if(a>0&&b<0||a>1) then {a=x+10;} else {b=x+10;}

X1 : if a > 0   falg=true else flag=false
 if flag(true) goto x2
 if flag(false) goto L1
X2 : if b < 0   flag=true else flag=false
 if flag(true) goto L0
 if flag(false) goto x3
X3 : if a > 1   flag=true else flag=false
 if flag(true) goto  L0
 if flag(false) goto L1
L0 :
t0 = x + 10
a = t0
goto L2
L1:
t1 = x + 10
b = t1
L2:
[cs6c144@localhost cs6c144]$ 
```