CASE WESTERN RESERVE UNIVERSITY
Case School of Engineering
Department of Electrical Engineering and Computer Science
EECS 337 Systems Programming (Compiler Design)
Fall 2013
Assignment #9
30 Points + 5 points extra credit
Due: November 5, 2013

Part 1: Reading
Compilers - Principles, Techniques, & Tools, 2nd Edition, Sections 7.1, 7.2, 7.4
Homework Exercises – laboratory assignment and answer the questions at the end of the assignment.

Introduction
In this assignment you copy and modify the previous calculator program to support C style if-else statements and add the equality and relational operators (==, !=, <, >, <=, >=). You design and implement a quad function to support the three-address code highlighted below. An if-else expression without an equality operator assumes the not equal to zero operation. For example, `if(a)` is the same as `if(a != 0)` and generates the IFTRUE quad.

1) X = Y op Z
2) X = op Y
3) X = Y
4) GOTO L
5) **IFTRUE X GOTO L** and **IFFALSE X GOTO L**
6) IF X relop Y GOTO L
7) CALL P,N or Y = CALL P,N and RETURN or RETURN Y
8) X = Y[i] or X[i] = Y
9) X = &Y or X = *Y or *X = Y

The scanner is updated for the new tokens and the parser is expanded to handle the C style if-else statements and the equality and relational operators. The quad.c and yystype.h files are updated to support the new quad function:

```
QUAD   *new_quad5( int operator, QUAD *q1, QUAD *q2, QUAD *q3)
```

For extra credit you add the not operator `(!)`. If the last operator on a linked-list is the not operator then modified it using back patching to creates the IFFALSE quad instead. For example: `if(!a)` will now generate the IFFALSE quad. You change the quad from a unary quad (2) into an IFFALSE quad (5). This optimization saves allocating one new quad on the linked list.

The equality and relational operators are part of an expression and do not have to be the last operation. For example: `if((a == 1) ^ (b != 0)) a = 1; else b = 1;` This C fragment creates the following sequence of three-address quads:

```
        t1 = a
        t2 = 1
        t3 = t1 == t2
        t4 = b
        t5 = 0
        t6 = t4 != t5
        t7 = t3 ^ t6
        IFTRUE t7 GOTO label1
        t8 = 1
        a = t8
        GOTO label2
label1:
        t9 = 1
        b = t9
label2:
```

Notice the assembler style format with labels in the first column and quads tabbed over. The equality and relational operators are treated as binary operators with a result of zero or one.

Precedence and Associatively of C Operators

OPERATORS	ASSOCIATIVITY
`() {} -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

The calculator program uses the yacc precedence operators (`%left, %right, %prec`) where the ansi C compiler uses productions to determine precedence. An ansi C compiler fragment for expressions is shown below. Notice how they implement part of the precedence table.

```
multiplicative_expr
        : cast_expr
        | multiplicative_expr '*' cast_expr
        | multiplicative_expr '/' cast_expr
        | multiplicative_expr '%' cast_expr
        ;
additive_expr
        : multiplicative_expr
        | additive_expr '+' multiplicative_expr
        | additive_expr '-' multiplicative_expr
        ;
shift_expr
        : additive_expr
        | shift_expr LEFT_OP additive_expr
        | shift_expr RIGHT_OP additive_expr
        ;
relational_expr
        : shift_expr
        | relational_expr '<' shift_expr
        | relational_expr '>' shift_expr
        | relational_expr LE_OP shift_expr
        | relational_expr GE_OP shift_expr
        ;
equality_expr
        : relational_expr
        | equality_expr EQ_OP relational_expr
        | equality_expr NE_OP relational_expr
        ;
and_expr
        : equality_expr
        | and_expr '&' equality_expr
        ;
```

Part 2: Laboratory

From a console window, make a directory on your computer in your EECS337 directory under your Case ID and call it hw09.

mkdir ~/EECS337/caseid/hw09/ ; where caseid is YOUR Case ID, enter all in lower case

Change directory to the hw09 directory.

cd ~/EECS337/caseid/hw09/

Download a copy of: hw09_caseid.tar file to the hw09 directory from http://blackboard.case.edu/ in the EECS337 homework assignment area. To untar the tar file type the command:

tar xvf hw09_caseid.tar

The following files will be created in the current working directory.

hw09_test.sh
Makefile
math21.txt
math22.txt
math23.txt // extra credit

Copy the following files from the assignment 08 directory to this directory with the commands:

```
cp ../hw08/lex.l .
cp ../hw08/main.c .
cp ../hw08/quad.c .
cp ../hw08/symbol_table.c .
cp ../hw08/yacc.y .
cp ../hw08/yystype.h .
```

You are now ready to solve the laboratory assignment.

Part 3: Laboratory Assignment

There are no changes to the main.c and symbol_table.c files for this assignment. You may want to go in and change the comments in the headers to note this is assignment 09.

Edit the yystype.h file and below in the external functions section add the function prototype for the new parser action.

```
/*
 *     new function for IF ELSE productions
 */
extern QUAD   *new_quad5( int operator, QUAD *q1, QUAD *q2, QUAD *q3);
```

Save the yystype.h file.

Edit the lex.l file and add two regular expressions below "$" and actions to support the if-else keywords.

```
"if"                    { return IF; }
"else"                  { return ELSE; }
```

Below the constant expressions and before the white space expression, add the relational operators. Notice the less than and greater than operators (<, >) are already defined.

```
"<="                    { return( LE); }
">="                    { return( GE); }
"=="                    { return( EQ); }
"!="                    { return( NE); }
```

Save the lex.l file.

Edit the yacc.y file and add the tokens shown below. The parser grammar uses six new tokens and the next four are used as quad operators.

```
%token IF
%token ELSE
%token LE
%token GE
%token EQ
%token NE
/* quads operators */
%token LABEL
%token GOTO
%token IFTRUE
%token IFFALSE
```

Insert the new precedence operators. Use the precedence table to determine the correct location.

```
%left EQ NE
%left '<' '>' GE LE
```

Remove the previous statement productions and insert the code below. These statements are the if-else and assignment statements. Notice the assignment statement now ends with a semi-colon and the if-else productions do not. This allows generating quads from C program fragments.

```
stmts  : IF '(' expr ')' stmts
       {
         $$.quad = new_quad5( IFTRUE, $3.quad, $5.quad, 0);
       }
       | IF '(' expr ')' stmts ELSE stmts
       {
              $$.quad = new_quad5( IFTRUE, $3.quad, $5.quad, $7.quad);
       }
       | ident '=' expr ';'
       {
              $$.quad = new_quad3( '=', $1.index, $3.quad);
       }
       ;
```

In the expression productions add the equality and relative operators using the binary quad.

```
/*     add relational operators */
       | expr LE expr
       {
              $$.quad = new_quad1( LE, $1.quad, $3.quad);
       }
       | expr GE expr
       {
              $$.quad = new_quad1( GE, $1.quad, $3.quad);
       }
       | expr EQ expr
       {
              $$.quad = new_quad1( EQ, $1.quad, $3.quad);
       }
       | expr NE expr
       {
              $$.quad = new_quad1( NE, $1.quad, $3.quad);
       }
       | expr '<' expr
       {
              $$.quad = new_quad1( '<', $1.quad, $3.quad);
       }
       | expr '>' expr
       {
              $$.quad = new_quad1( '>', $1.quad, $3.quad);
       }
```

Save the yacc.y file.

Edit the quad.c file and before the print_quad function insert the relational operators strings.

```
/*
 *     define the relational operator strings
 */
char   *relational[] =
{
       "<=",
       ">=",
       "==",
       "!=",
};
```

Inside the print_quad function add the single character relational operators to the binary operators and include the print tab function. This reserves the first column for labels statements.

```
       case '<':
       case '>':
            printf( "\t");
```

After the break and before the case UMINUS, add the two character relational operators. Notice the single operator values to represent the double character operators.

```
       case LE:
       case GE:
       case EQ:
       case NE:
            printf( "\t");
            print_quad_operand( quad->dst_type, quad->dst_index);
            printf( " = ");
            print_quad_operand( quad->op1_type, quad->op1_index);
            printf( " %s ", relational[ quad->operator - LE]);
            print_quad_operand( quad->op2_type, quad->op2_index);
            break;
```

Insert the print tab function into the other three case statements. This prints each quad away from the left side. This is an old assembler language rule.

```
            printf( "\t");
```

Add to the print_quad function the new code to support IFTRUE, IFFALSE, LABEL and GOTO operators.

```
       case IFTRUE:
            printf( "\t");
            printf( "IFTRUE ");
            print_quad_operand( quad->dst_type, quad->dst_index);
            printf( " GOTO ");
            print_quad_operand( quad->op1_type, quad->op1_index);
            break;
       case IFFALSE:
            printf( "\t");
            printf( "IFFALSE ");
            print_quad_operand( quad->dst_type, quad->dst_index);
            printf( " GOTO ");
            print_quad_operand( quad->op1_type, quad->op1_index);
            break;
       case LABEL:
            print_quad_operand( quad->dst_type, quad->dst_index);
            printf( ": ");
            break;
       case GOTO:
            printf( "\t");
            printf( "GOTO ");
            print_quad_operand( quad->dst_type, quad->dst_index);
            break;
```

At the bottom of the file add the new quad function. Design and implement this function.

```
/*
 *     allocate a quad5 function
               $$.quad = new_quad5( IFTRUE, $3.quad, $5.quad, 0);
               $$.quad = new_quad5( IFTRUE, $3.quad, $5.quad, $7.quad);
 */
QUAD   *new_quad5( int operator, QUAD *q1, QUAD *q2, QUAD *q3)
{
       return q1;
}
```

Save the quad.c file.

Build the calc program and fix any errors using the commands:
make clean
make

To test your version, and print the symbol table type the command line:
./calc +symbol and at prompt enter an expression: **if(a3 ^ 0x01) a3 = a3 + b2; else a3 = 0x01;**
To exit the application type on the command line: **$**
To test with the symbol table and with the yydebug statements type:
./calc +symbol +yydebug and enter a math expression as before.

Your output for the example above should be:

```
for caseid start time: Fri Oct 25 14:17:20 2013
Enter calculator expression and $ to exit
if( a3 ^ 0x01) a3 = a3 + b2; else a3 = 0x01;
        t1 = a3
        t2 = 0x01
        t3 = t1 ^ t2
        IFTRUE t3 GOTO label1
        t4 = a3
        t5 = b2
        t6 = t4 + t5
        a3 = t6
        GOTO label2
label1:
        t7 = 0x01
        a3 = t7
label2:
$
symbol table:
index: 1 identifier: a3 length: 3
index: 2 constant: 0x01 length: 5 format: hexadecimal
index: 3 identifier: b2 length: 3
```

Test using the test files using the commands below:
./calc math21.txt
./calc math22.txt

Part 4: Output Generation
When all your lab assignments have been completed execute the homework script file "./hw09_test.sh" using the command below.
./hw09_test.sh &> hw09_test.txt

Print out the hw09_test.txt file and put your name, assignment number and date on it. Turn in the printout and answer the questions at the end of the assignment.

Your final directory structure for the calc compiler should be as below (using your Case ID):
EECS337/caseid/hw09/Makefile
EECS337/caseid/hw09/calc
EECS337/caseid/hw09/hw09_caseid.tar
EECS337/caseid/hw09/hw09_test.sh
EECS337/caseid/hw09/hw09_test.txt
EECS337/caseid/hw09/lex.l
EECS337/caseid/hw09/main.c
EECS337/caseid/hw09/math21.txt
EECS337/caseid/hw09/math22.txt
EECS337/caseid/hw09/math23.txt
EECS337/caseid/hw09/quad.c
EECS337/caseid/hw09/symbol.c
EECS337/caseid/hw09/yacc.y
EECS337/caseid/hw09/yystype.h

Part 5: Extra Credit (5 points)
Save a copy of your code and edit the yacc .y file. Add the not (!) operator to the precedence section. Use the precedence table to determine the correct location. Then add the new not production to the expression productions. This uses the same quad function as the other unary operators. Notice this operator is already defined in the lex.l file.

```
        | '!' expr
        {
                $$.quad = new_quad2( '!', $2.quad);
        }
```

Save the yacc.y file.

Edit the quad.c file. Change the print_quad function to handle the not operator. For example add code to the ~ operator and use the quad operator to print the character.

```
        case '~' :
        case '!' :
             printf( "\t");
             print_quad_operand( quad->dst_type, quad->dst_index);
             printf( " = %c ", quad->operator);
             print_quad_operand( quad->op1_type, quad->op1_index);
             break;
```

Change the new_quad5 function to look for the not operator on the last quad and back patch it to an IFFALSE quad. In this case you need to move operand 1 to the destination and set operand 1 to a new label. This last quad now links to quad 2 without having to call the new_quad function.

```
                quad1 = end_quad_list( q1);
/*
 *      backpatch ! quad into iffalse quad
 */
                if( quad1->operator == '!')
```

Save the quad.c file.

Edit the hw09_test.sh file and uncomment the line to test with math23.txt and save the file.

When the extra credit code is complete and working then execute the homework script file "./hw09_test.sh" using the command below.
./hw09_test.sh &> hw09_test.txt

Print out this hw09_test.txt file instead of the file from Part 4. Put your name, assignment number, date on it and mark it with EXTRA CREDIT. Turn in this printout and answer the questions at the end of the assignment.

Part 6: Laboratory Questions

1) When you have a syntax error "a = 1" (left off semi-colon) and exit the program ($), why is there a memory leak error? (*Error: memory leak: 40)*

2) When getting the last destination type and index off an expression (expr) quad linked list, why is it always a temporary operand and never a label quad?

3) Why are unconditional jumps (GOTO L) generated as part of the if-else statement, which use conditional jump three address codes (IFTRUE X GOTO L)?

4) Why does this grammar now have a shift/reduce error? Hint use yacc –v yacc.y and look at the y.output file.

5) The relational operators are added to parser using which quad function?

6) In the scanner, why do you put the if-else keywords before the identifier expression?

7) Why does data.temp get reset at the end of each new line where data.label does not?

8) Explain a method to optimize or reduce the number of quads generated in the assignment?

9) Re-write the quads generated for: **if(a3 ^ 0x01) a3 = a3 + b2; else a3 = 0x01;** using optimized quads.

10) Write PIC 16F84A assembler code to implement the quads from the question above.