CASE WESTERN RESERVE UNIVERSITY
Case School of Engineering
Department of Electrical Engineering and Computer Science
EECS 337 Systems Programming (Compiler Design)
Fall 2013
Assignment #8
30 Points + 5 points extra credit
Due: October 29, 2013

Part 1: Reading
Compilers - Principles, Techniques, & Tools, 2nd Edition, Sections 6.6, 6.7
Homework Exercises – laboratory assignment and answer the questions at the end of the assignment.

Introduction
In this assignment, you implement three-address code (Quadruples or QUAD) for the calculator program. In the previous two assignments you implemented an interpreter and calculated the final result in real time. In this assignment the interpreter actions are removed and new code is inserted to generate a quad linked-list (dynamic memory). A new production (stmts) is added to the starting production to create a single linked-list. The linked-list is printed to the screen and the memory is freed. In the last assignment identifiers were added to the symbol table and constants were encoded and returned to the parser. In this assignment the scanner installs the identifiers and constants into the symbol table and return the index to the parser. The constants do not need to be encoded in this case because they are only printed to the screen.

You write C code (quad.c) to implement the parser functions to support the first three quad definitions:

1)	X = Y op Z	binary	$$.quad = new_quad1('+', $1.quad, $3.quad);
2)	X = op Y	unary	$$.quad = new_quad2(UMINUS, $2.quad);
3)	X = Y	copy	$$.quad = new_quad3('=', $1.index, $3.quad);

New Calculator Operations
The operations of the new calculator program are to generate a linked list and print the list to the screen. Use the symbol table flag to print the symbol table when the program exits. For example:
./calc +symbol
for caseid start time: Tue Oct 15 09:32:59 2013
Enter calculator expression and $ to exit
2
t1 = 2
a = 2
t1 = 2
a = t1
b = a * 3
t1 = a
t2 = 3
t3 = t1 * t2
b = t3
$
symbol table:
index: 1 constant: 2 length: 2 format: decimal
index: 2 identifier: a length: 2
index: 3 identifier: b length: 2
index: 4 constant: 3 length: 2 format: decimal

When you enter **2** the QUAD copy function (3) is called and a quad is generated with the constant symbol table index. It is pushed onto the stack and in the last parser reduction the quad is printed to the screen, the dynamic memory is freed and the temporary index is reset. When you enter **a = 2** the same QUAD copy function (3) is called but includes the identifier on the left (index) and the expression (quad) on the right. When you enter **b = a * 3** the QUAD binary function (1) is called with two linked-lists. The operands are the destinations on the ends of the linked list. When you exit, the symbol table is printed to the screen showing the identifiers and constants.

Part 2: Laboratory
From a console window, make a directory on your computer in your EECS337 directory under your Case ID and call it hw08.
mkdir ~/EECS337/caseid/hw08/ ; where caseid is YOUR Case ID, enter all in lower case

Change directory to the hw08 directory.
cd ~/EECS337/caseid/hw08/

Download a copy of hw08_caseid.tar file to the hw08 directory from http://blackboard.case.edu/ in the EECS337 homework assignment area. To untar the tar file type the command.
tar xvf hw08_caseid.tar

The following files will be created in the current working directory.
Makefile
hw08_test.sh
main.c
math20.txt
quad.c
symbol_table.c
yystype.h

Copy the previous files (first.l, second.y) to the current working directory and change the names to lex.l and yacc.y.
cp ../hw07/first.l lex.l
cp ../hw07/second.y yacc.y

You are now ready to solve the laboratory assignment.

Part 3: Laboratory Assignment
Edit the main.c file and change caseid to your Case ID. This is a modified version from assignment 04 and has the global data structure, command line flags (+yydebug, +symbol, +debug) and the initialize and exit functions. Look at the exit function and see where it prints the symbol table and checks for memory leaks. Save the main.c file.

Edit the yystype.h file and change caseid to your Case ID. Look at the data structures SYMBOL_TABLE, QUAD, YYSTYPE and DATA. We want to install the identifiers and constants into the symbol table and return an index from the static symbol table. In this case we want to use an integer value in the QUAD structure. The SYMBOL_TABLE uses static fields for strings and number of symbol table entries. We define types for identifiers, constants, temporary and labels. The YYSTYPE structure is a defined as a structure with two fields (index, QUAD*). The DATA structure has new fields (memory, temp, label) used in the quad generation. Memory keeps track of the amount of dynamic memory allocated, temp is a field to create temporary operands (t1, t2) and label is a field to create labels (label1, label2). If you change the structure definitions then change your source code to match. Save the yystype.h file.

Edit the lex.l file and update the file name to lex.l and edit history comment. Remove the previous symbol table declarations and identifier function and replace it with the line.
```
#include "symbol_table.c"
```

Insert the new install function. This is the same function you have written before. It searches the symbol table for the same string and if found returns the index. Otherwise it creates a new entry.

```
/*
 *     scanner symbol table register function
 *     return an index into the symbol table (static)
 */
int    install( int type, char *buffer, unsigned int length, int format)
{
       int    index;
/*
 *     find the same string and return the index
 */
       for( index = 1; index <= data.index; index++)
            if( data.st[ index].type == type && data.st[ index].length == length
&& !strcmp( data.st[ index].buffer, buffer))
              return index;
/*
 *     else update to the next symbol table field
 */
       data.index++;
       data.st[ data.index].type = type;
       strcpy( data.st[ data.index].buffer, buffer);
       data.st[ data.index].length = length;   // includes the null
       data.st[ data.index].format = format;
       return data.index;
};
```

Change all the regular expressions to use the new install function. Do this for both IDENTIFERS and CONSTANTS.

```
{L}({L}|{D})*       { yylval.index = install( TYPE_IDENTIFIER, yytext, yyleng +
1, FORMAT_NONE); return( IDENTIFIER); }
0[xX]{H}+{IS}?             { yylval.index = install( TYPE_CONSTANT, yytext,
yyleng + 1, FORMAT_HEXADECIMAL); return( CONSTANT); }
```

Save the lex.l file.

Edit the yacc.y file and update the file name to yacc.y and edit history comment. Remove the print_yystype function. Remove the two tokens INTEGER and FLOAT and replace them with CONSTANT. Replace the lines production with the code below.

```
lines  : lines stmts '\n'
       {
/*
 *     print the quad list
 */
              print_quad_list( $2.quad);
/*
 *     free the quad list
 */
              free_quad_list( $2.quad);
       }
       | lines error '\n' { yyerror(" reenter previous line: "); yyerrok; }
       | /* empty */
       ;
stmts  : expr
       | ident '=' expr
       {
              $$.quad = new_quad3( '=', $1.index, $3.quad);
       }
       ;
```

The new production (stmts) allows the linked list to be handled at one location. Now remove all the previous parser actions and replace them with a call to the quad generation functions. Edit the number production and replace INTEGER and FLOAT with CONSTANT. Save the yacc.y file.

Edit the quad.c file. The routines to dynamically allocate and free QUAD structures are provided. There are other routines to print the QUAD data structures, allocate temporaries and labels. You need to design and implement the three parser functions to allocate a quad linked list for the three supported quad types.

A) The new_quad3 function (X = Y) receives from the parser the operator ('=') and the index into the symbol table and possibly an expression linked list. It generates a new quad and links it to the linked list and returns the top of the list to the parser.

```
QUAD   *new_quad3( int operator, int index, QUAD *q1)
```

Use the index into the symbol table to determine the type of the operand. Use the new_quad function to allocate the quad and pass parameters to the function to initialize the fields. A temporary operand is created for the destination for the identifier and constant productions. The assignment production receives a quad linked list on the right hand side. In that case, use the end_quad_list function to find the last quad (quad1) to extract the type and index fields. Below are two examples for allocating quads for identifiers and constants or the assignment operation.

```
quad = new_quad( operator, TYPE_TEMPORARY, next_temp(), data.st[ index].type,
index, 0, 0);
quad = new_quad( operator, data.st[ index].type, index, quad1->dst_type, quad1-
>dst_index, 0, 0);
```

B) The new_quad2 function (X = op Y) receives from the parser the unary operators (UMINUS, '~') and the expression quad list. Use the end_quad_list function to find the last quad to extract the Y operand type and index fields. Link the new quad to the end of the quad linked list and return the top of the quad linked list to the parser.

```
QUAD   *new_quad2( int operator, QUAD *q1)
```

C) The new_quad1 function (X = Y op Z) receives from the parser the binary operators and two quad linked lists. Use the end_quad_list function to find the last quad for both linked lists and use them to create the new quad. Link the end of the first quad linked list to the top of the second quad linked list and the new quad to the end of the second linked list.

```
QUAD   *new_quad1( int operator, QUAD *q1, QUAD *q2)
```

Implement these three parser functions. Your output should generate a list of quads and you should not have any memory leaks when the program exits. Save the quad.c file.

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

To test your version, on the command line type:
./calc +symbol

To exit the application type: **$**

To test with the yacc yydebug statements or both type:
./calc +yydebug
./calc +symbol +yydebug

Your output for the example **(2 + 3) * (a + b)** should be:

./calc +symbol
for caseid start time: Tue Oct 15 10:28:43 2013
Enter calculator expression and $ to exit
(2+3)*(a+b)
t1 = 2
t2 = 3
t3 = t1 + t2
t4 = a
t5 = b
t6 = t4 + t5
*t7 = t3 * t6*
$
symbol table:
index: 1 constant: 2 length: 2 format: decimal
index: 2 constant: 3 length: 2 format: decimal
index: 3 identifier: a length: 2
index: 4 identifier: b length: 2

Test using the previous assignment test files using the command below:
./calc +symbol < ../hw07/math18.txt
./calc +symbol < ../hw07/math19.txt

Part 4: Output Generation
When all your lab assignments have been completed execute the homework script file "./hw08_test.sh". The script file shall remove the old object files, make all source programs and run the tests. To redirect the standard error (stderr) and standard output (stdout) to a file use the following command. The files from assignment should also work.
./hw08_test.sh &> hw08_test.txt

Print out the hw08_test.txt file and put your name, assignment number and date on it. Turn in the files for the assignment 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/hw08/Makefile
EECS337/caseid/hw08/calc
EECS337/caseid/hw08/hw08_caseid.tar
EECS337/caseid/hw08/hw08_test.sh
EECS337/caseid/hw08/hw08_test.txt
EECS337/caseid/hw08/lex.l
EECS337/caseid/hw08/main.c
EECS337/caseid/hw08/math20.txt
EECS337/caseid/hw08/quad.c
EECS337/caseid/hw08/yacc.y
EECS337/caseid/hw08/yystype.h

Part 5: Extra Credit (5 points)
Edit the source files and add support for the unary + operator (UPLUS). In the yacc.y file, you will want to add a UPLUS terminal symbol to the %right precedence line and add a production to handle the + unary operator (| '+' expr %prec UPLUS). Then update the print_quad function in the calc.c file to handle the new operator. To test your new version type:
./calc +symbol

Enter the math expression: **(+2 + +3) * (+a + +b)**

Your output should be:
for caseid start time: Wed Oct 16 14:26:19 2013
Enter calculator expression and $ to exit
(+2 + +3) * (+a + +b)
t1 = 2
t2 = + t1
t3 = 3
t4 = + t3
t5 = t2 + t4
t6 = a
t7 = + t6
t8 = b
t9 = + t8
t10 = t7 + t9
*t11 = t5 * t10*
$
symbol table:
index: 1 constant: 2 length: 2 format: decimal
index: 2 constant: 3 length: 2 format: decimal
index: 3 identifier: a length: 2
index: 4 identifier: b length: 2

Test your calculator compiler using the extra credit math20.txt file by typing the command:
./calc < math20.txt

Edit the hw08_test.sh file and remove the comment for testing math20.txt. Save the script file. When the extra credit code is complete and working then execute the homework script file "./hw08_test.sh" using the command below.
./hw08_test.sh &> hw08_test.txt

Print out this version of the **hw08_test.txt** output file. Mark this output with **extra credit** and put your name, assignment number, date on it and turn it in instead of the output file from Part 4. Answer the questions at the end of the assignment.

Part 6: Laboratory Questions

On the next page, answer the question and write the three-address code (quads) for the C code fragments. Then convert by hand the quads into PIC assembler code for the PIC18F84A processor. You need to support the quads highlighted below.
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

Assignment 08 Convert C code into QUADS and PIC assembler code.

1) Why define another set of types (TYPE_IDENTIFIER, TYPE_CONSTANT) in the symbol table structure instead of using the yacc tokens (IDENTIFIER, CONSTANT), which are already defined?

Convert the following code fragments into QUADs and PIC code.

2) `char a = 9; char b = 3 - a;`

3) `char a; char b[ 10]; b[4] = 2; a = b[4];`

4) `char a = 0; if ( a == 0) a = a + 2; else a = a - 1;`