CASE WESTERN RESERVE UNIVERSITY
Case School of Engineering
Department of Electrical Engineering and Computer Science
EECS 337 Systems Programming (Compiler Design)
Fall 2013
Assignment #10
30 Points + 5 points extra credit
Due: November 12, 2013

Part 1: Reading
Compilers - Principles, Techniques, & Tools, 2nd Edition, Sections 7.5, 7.6, 8.1
Homework Exercises – problems 8.2.1 (a,b,c,d), 8.2.2 (a) using PIC 16F84 instructions, the laboratory assignment and answer the questions at the end of the assignment.

Introduction
In this assignment you expand the calculator program to support C style declaration statements (char, short, int, long) and implement code for handling single dimension arrays X = Y[Z] as part of expressions on the right side. For extra credit you add code to handle single dimension arrays X[Y] = Z on the left side. You add a new C function to support the highlighted quad (8) below. You copy and modify the previous version to support the new features.

1) X = Y op Z
2) X = op Y
3) X = Y
4) GOTO L or 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[Z] or **X[Y] = Z**
9) X = &Y or X = *Y or *X = Y

The C programming language has a number of type specifiers. These break down into elementary and aggregate data types. The data type can also be modified with a storage classifier. The list below shows the type and storage class specifiers from the ansi C compiler.

```
type_specifier
       : CHAR                          // elementary 1 bytes
       | SHORT                         // elementary 2 bytes
       | INT                           // elementary 4 bytes
       | LONG                          // elementary 4 or 8 bytes
       | SIGNED                        // elementary 4 bytes
       | UNSIGNED                      // elementary 4 bytes
       | FLOAT                         // elementary 4 bytes
       | DOUBLE                        // elementary 8 bytes
       | CONST                         // elementary 4 bytes
       | VOLATILE                      // elementary 4 bytes
       | VOID                          // elementary 1 bytes not variable
       | struct_or_union_specifier     // aggregate n bytes
       | enum_specifier                // aggregate n bytes
       | TYPE_NAME                     // elementary or aggregate type
       ;
storage_class_specifier
       : TYPEDEF                       // elementary or aggregate type
       | EXTERN                        // elementary 4 bytes
       | STATIC                        // elementary 4 bytes
       | AUTO                          // elementary 4 bytes
       | REGISTER                      // elementary 4 bytes
       ;
```

Declarations and Array Indexing

We would like to add C style declarations for elementary and single dimension array types into the calculator program. We add char, short, int and long declaration types to the grammar for elementary variables and single dimension arrays. Most computer architectures assume 8 bits per byte with elementary variable sizes defined in powers of 2s (1, 2, 4, 8, 16). Indexing into a character array is then straightforward. For example the following C code fragment with an elementary character variable and array generates the following quads.

```
char   c,
char   c1[ 16];
c = c1[ 15];

       t1 = 15
       t2 = c1 [ t1 ]
       c = t2
```

Where the following C code fragment using short declarations generates the following quads. Notice the extra quad to multiple the previous destination by the sizeof the type specifier in order to determine the correct array index value. The same is true for integers and long data types.

```
short  s;
short  s2[ 16];
s = s2[ 15];

       t1 = 15
       t2 = t1 * 2
       t3 = s2 [ t2 ]
       s = t3
```

The following C code declarations with integers and longs generate the following quads.

```
int    i;
int    i4[ 16];
long   l;
long   l8[ 16];
i = i4[ 15];
l = l8[ 15];

       t1 = 15
       t2 = t1 * 4
       t3 = i4 [ t2 ]
       i = t3
       t1 = 15
       t2 = t1 * 8
       t3 = l8 [ t2 ]
       l = t3
```

When the array reference is on the left size the same set of operations are used to determine the correct array index value.

```
c1[ 15] = c;
s2[ 15] = s;

       t1 = 15
       t2 = c
       c1 [ t1 ] = t2
       t1 = 15
       t3 = t1 * 2
       t2 = s
       s2 [ t3 ] = t2
```

Part 2: Laboratory
From a console window, make a directory on your computer in your EECS337 directory under your Case ID and call it hw10.
mkdir ~/EECS337/caseid/hw10/ ; where caseid is YOUR Case ID, enter all in lower case

Change directory to the hw10 directory.
cd ~/EECS337/caseid/hw10/

Download a copy of: hw10_caseid.tar file to the hw10 directory from http://blackboard.case.edu/ in the EECS337 homework assignment area. To untar the tar file type the command:
tar xvf hw10_caseid.tar

The following files will be created in the current working directory.
Makefile
hw10_test.sh
math24.txt
math25.txt
math26.txt // extra credit
symbol_table.c

Copy the following files from the assignment 09 directory to this directory with the commands:
cp ../hw09/lex.l .
cp ../hw09/main.c .
cp ../hw09/quad.c .
cp ../hw09/yacc.y .
cp ../hw09/yystype.h .

Optional: If you did not complete the last assignment you can download the hw09solutions.tar file from the assignment directory and use that as your starting code. In this case edit all the files and change "caseid" to your Case ID. Use the command "tar xvf hw09_solutions.tar" and the hw09/ directory will be created with the source files.

Look at the new symbol_table.c file. This file has been update to print the symbol table type specifiers and the array variables. A new_symbol function has been added to insert new symbol declarations into the symbol table. All variables are assumed to be at level 0 and normally defined in the static or global memory section. If you are using your own symbol table file then update your version of the file. In this case use the command below to copy your previous version to the current working directory and update the file to generate an equivalent symbol table output.
cp ../hw09/symbol_table .

You are now ready to solve the laboratory assignment.

Part 3: Laboratory Assignment
Edit the yystype.h file. Inside the symbol table data structure insert the code below. Notice the state assignment values are based on the sizeof the type specifiers. For the calculator program we only support declarations of these four elementary types.

```
#define      SPECIFIER_CHAR            1
#define      SPECIFIER_SHORT           2
#define      SPECIFIER_INT             4
#define      SPECIFIER_LONG            8
       int   specifier;
       int   sizeofspecifier;
       int   sizeofarray;
```

All constants are stored in the symbol table so the values (2, 4, 8) must also be installed. The index for the type specifier constants are stored in the sizeofspecifier variable for that identifier.

For C style array declarations (int a[8];) the constant is automatically installed by the scanner (lex.l) and the index into the symbol table is stored in the sizeofarray variable. For scalar variables the sizeofarray is zero. Inside the DATA structure add the errors variable after the label variable.

```
	int	errors;
```

After the extern new_quad5 function add the two new extern function definitions.

```
extern QUAD   *new_quad8( int operator, int index, QUAD *q1, QUAD *q2);
extern int    new_symbol( int specifier, int identifier, int constant);
```

Save the yystype.h file.

Edit the main.c file and find the main_exit function. Replace the `return 0;` statement with the code below. We now start to do static checking so report the number of total compiler errors.

```
/*
 *	check for compiler errors
 */
	if( data.errors)
		fprintf( stderr, "Errors: %d\n", data.errors);
	return data.errors;
```

Save the main.c file.

Edit the lex.l file and add the keywords and actions for the type declarations. Then save the file.

```
"char"			{ yylval.index = CHAR; return CHAR; }
"short"			{ yylval.index = SHORT; return SHORT; }
"int"			{ yylval.index = INT; return INT; }
"long"			{ yylval.index = LONG; return LONG; }
```

Edit the yacc.y file and after the IDENTIFIER token insert the four new parser tokens shown below. The tokens define the scalar types allowed in the declaration productions.

```
%token CHAR
%token SHORT
%token INT
%token LONG
```

Insert the declarations line below into the lines production before the error production.

```
	| lines decls '\n'
```

After the lines semi-colon and before the statements production add the declaration productions. Notice these functions do not return any values and only modify the symbol table.

```
decls	: type ident ';'
	{
		new_symbol( $1.index, $2.index, 0);
	}
	| type ident '[' number ']' ';'
	{
		new_symbol( $1.index, $2.index, $4.index);
	}
	;
type	: CHAR
	| SHORT
	| INT
	| LONG
	;
```

At the end of the expression productions and before the semi-colon add the right side identifier array production shown below.

```
        | ident '[' expr ']'
        {
                $$.quad = new_quad8( ']', $1.index, $3.quad, 0);
        }
```

Save the yacc.y file.

Edit the quad.c file. Inside the print_quad function add the code to print the right side of quad 8.

```
        case ']':
             printf( "\t");
             print_quad_operand( quad->dst_type, quad->dst_index);
             printf( " = ");
             print_quad_operand( quad->op1_type, quad->op1_index);
             printf( " [ ");
             print_quad_operand( quad->op2_type, quad->op2_index);
             printf( " ] ");
             break;
```

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

```
/*
 *     allocate a quad8 function
 *             $$.quad = new_quad8( ']', $1.index, $3.quad, 0);
 */
QUAD   *new_quad8( int operator, int index, QUAD *q1, QUAD *q2)
{
        return q1;
}
```

Save the quad.c file.

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

To test your version, type:
./calc +symbol and enter the lines below.

```
int a;
int b[ 10];
a = b[a+1];
        t1 = a
        t2 = 1
        t3 = t1 + t2
        t4 = t3 * 4
        t5 = b [ t4 ]
        a = t5
$
symbol table:
index: 1 identifier: a length: 2 specifier: int
index: 2 constant: 4 length: 2 format: decimal
index: 3 identifier: b[10] length: 2 specifier: int
index: 4 constant: 10 length: 3 format: decimal
index: 5 constant: 1 length: 2 format: decimal
```

Test using the test files using the commands below:
./calc math24.txt
./calc math25.txt

Part 4: Output Generation
When all your lab assignments have been completed execute the homework script file "./hw10_test.sh" using the command below.
./hw10_test.sh &> hw10_test.txt

Print out the hw10_test.txt file and put your name, assignment number and date on it. Turn in the file for the assignment and answer the questions at the end of the assignment.

Part 5: Extra Credit (5 points)
Save a copy of your code and then edit the following two files. The code is used to support assignments with array identifiers on the left size.

Edit the yacc.y file and add the production shown below, after the assignment production inside the statements (stmts) production and before the semi-colon. This supports the quad X[Y] = Z.

```
| ident '[' expr ']' '=' expr ';'
{
      $$.quad = new_quad8( '[', $1.index, $3.quad, $6.quad);
}
```

Save the yacc.y file.

Edit the quad.c file and add to the print_quad function the code below. This adds support to print the quad, which is defined above.

```
case '[':
     printf( "\t");
     print_quad_operand( quad->dst_type, quad->dst_index);
     printf( " [ ");
     print_quad_operand( quad->op1_type, quad->op1_index);
     printf( " ] = ");
     print_quad_operand( quad->op2_type, quad->op2_index);
     break;
```

Edit the new_quad8 function to support this new operation. Save the quad.c file.

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

Test using the extra credit test file using the command below:
./calc math26.txt

Edit the hw10_test.sh file and remove the comment to test the math26.txt file. When the extra credit code is complete and working then execute the homework script file "./hw10_test.sh" using the command below.
./hw10_test.sh &> hw10_test.txt

Print out this hw10_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) In symbol_table.c, why does the char *types[] array have a bunch of empty strings?

2) What error is generated when using a floating-point constant for declaring an array size? Example: `int b[1.1];`

3) In the function new_symbol, why are the sizeof the type specifiers (2, 4, 8) installed into the symbol table?

4) Can you use a storage class type to declare a variable? Example: `extern extern1;`

5) Can you use the sizeof() operator on a storage class type? Example: `sizeof( extern)`