CASE WESTERN RESERVE UNIVERSITY
Case School of Engineering
Department of Electrical Engineering and Computer Science
EECS 337 Systems Programming (Compiler Design)
Fall 2013
Assignment #06
30 Points + 5 extra credit
Due: October 8, 2013

Part 1: Reading
Compilers - Principles, Techniques, & Tools, 2nd Edition, Sections 4.5, 4.6, 4.9
Homework Exercises – problem 4.5.3 (b) and laboratory assignment.

Introduction
An example of a math calculator program (interpreter) using lex (first.l) and yacc (second.y) is found on page 295-296 of your book. You implement and test the program using the commands:

```
lex first.l
yacc second.y
cc y.tab.c -ly -ll
./a.out
./a.out < math0.txt
```

The first command creates a lex.yy.c file. The second command creates a y.tab.c file. The third line compiles the y.tab.c file into an executable file (a.out). You run the executable program with the forth command and enter a string from the console. If you enter a string (1e1+1e2) and hit return, you will get an output result (110) printed to the screen. To exit the program you must use CTRL-C to terminal the program. The fifth line shows redirecting a file to the executable to test the program and terminates when it see the end of file (EOF) character.

The way the source code from the book is formatted, lex.yy.c file from the first line is included in the bottom of the yacc (second.y) file. This is why it is **not** included in the third line as a separate file to compile and link. If you change the first.l file you must execute the first three commands in sequence again. It is easier to use a make file (Makefile) to compile and link each file independently. We also would like to include a main program (calc.c) so we can initialize the compiler/interpreter before calling the yacc parser (yyparse).

In this assignment you change the implementation of the math operators (+ - * /) from using a double (8 bytes) type to using a long double (16 bytes) type. Then you add the bitwise operators (& | ^ % ~), which require integer types (long long). You update the YYSTYPE structure to handle integer and floating-point numbers. You will need to handle constant type conversion for the bitwise operators. For extra credit you remember the constant format (floating point, decimal, octal, hexadecimal) for correctly displaying the output result (0xff & 0x0f -> 0x0f).

For encoding integer and floating-point numbers you replace the (first.l) lex regular expression for number to use the expressions from the ansi_c compiler (shown below).

```
number [0-9]+\e.?|[0-9]*\e.[0-9]+
%%
{number}      { sscanf( yytext, "%lf", &yylval); return NUMBER; }

0[xX]{H}+{IS}?              // hex
0{D}+{IS}?                  // octal
{D}+{IS}?                   // decimal
{D}+{E}{FS}?                // float
{D}*"."{D}+({E})?{FS}?      // float
{D}+"."{D}*({E})?{FS}?      // float
```

Laboratory Assignment

In this assignment you encode the numbers using lex regular expressions and the C sscanf() function into a static YYSTYPE data structure. You update the current lex scanner to add lex regular expressions to encode integers (decimal, octal, hexadecimal) and floating-point numbers. You then expand the math operators (+ - * /) to include the bitwise instructions (| ^ & % ~) operators. You add action code to manually handle the operand types in the parser to maintain the correct operand type.

In the calculator program the parser uses the yacc precedence operators (%left %right %prec). The precedence statements are normally located in the top of the yacc file (second.y) just below the %token statements. The precedence is defined from lower to higher precedence going down the page. The binary operators are referenced to the left (a op b) and the unary operators are referenced to the right (op b). For example: (-2+3) where minus (unary) is applied first and then plus (binary) is used next ((-2)+3).

The book version of the calculator program supports double as the yylval data type and only returns floating point numbers (1e1). The parser is designed to handle a syntax error and to resynchronize at the end of line (‘\n’). Look at the second.y file and see “lines” is defined as an expression, error or empty line. The action yyerrok clears the error and allows the parser to continue processing the input stream. The version shown below is slightly different than the version shown on page 296 and removes the extra productions. We would like one place to handle the output result.

```
lines  : lines expr '\n' { printf("%g\n", $2); }
       | lines error '\n' { yyerror(" reenter previous line: "); yyerrok; }
       | /* empty */
       ;
```

In this assignment you redefined YYSTPE to using a structure definition to hold type information, INTEGER and FLOAT values. This changes the size of the yacc stack from a single scalar value to a larger structure definition. So while the size of the yacc stack is bigger in this implementation it does provide direct memory references for increased speed compared to dynamic address pointer implementations (indirect addresses). The math operators (op) already included are:

'+' addition
'-' subtraction
'*' multiple
'/' divide
'-' unary minus

The new logical bit-wise operators are:

'|' IOR
'^' XOR
'&' AND
'%' MOD
'~' NOT (unary)

You code the parser actions to support the correct data types. For the math instructions, if all the input operands are INTEGER then the result is an INTEGER. If any of the operands are a FLOAT then the result is a FLOAT. The data conversion methods are shown below for the math operators.

INTEGER = INTEGER op INTEGER
FLOAT = FLOAT op INTEGER
FLOAT = INTEGER op FLOAT
FLOAT = FLOAT op FLOAT
INTEGER = op INTEGER
FLOAT = op FLOAT

For the logical bit-wise operators all the FLOAT input operands need to be converted into INTEGER and the results is always an INTEGER. The data conversion methods are shown below for the bit-wise operators.

INTEGER = INTEGER op INTEGER
INTEGER = FLOAT op INTEGER
INTEGER = INTEGER op FLOAT
INTEGER = FLOAT op FLOAT
INTEGER = op INTEGER
INTEGER = op FLOAT

To achieve maximum execution speed the YYSTYPE data structure is defined using a static structure definition and the math and logic operations are coded directly in-line. The actions could be coded into just a few functions to optimize on space. The trade-off for increased speed is more source code to maintain and gives a larger executable image. This model allows yacc programs to be used in soft real time applications such as interpreters and processing stages in routers and communication systems.

In this assignment you shall implement the functions to encode constant values in the scanner. You should be able to support: integers (ddd), octal (0ooo), hexadecimal (0xhhh), and floating-point numbers (d.de-d). The current version defined in the code supports only floating-point numbers (math0.txt).

Part 2: Laboratory
From a console window, make a directory on your computer in your EECS337 directory under your Case ID and call it hw06.
mkdir ~/EECS337/caseid/hw06/ ; where caseid is YOUR Case ID, enter all in lower case

Change directory to the hw06 directory.
cd ~/EECS337/caseid/hw06/

Download a copy of: hw06_caseid.tar file to the hw06 directory from http://blackboard.case.edu/ from the EECS337 homework assignment area. To untar the tar file type the command:
tar xvf hw06_caseid.tar

The following files will be created in the current working directory.
book/book.sh
book/first.l
book/second.y
calc.c
first.l
hw06_test.sh
Makefile
math0.txt
math1.txt
math2.txt
math3.txt
math4.txt
math5.txt
math6.txt
math7.txt
math8.txt
math9.txt
second.y
yystype.h

The book subdirectory contains the original software from the book. In the current directory the software has been updated to use a make file, moves the YYSTYPE definition into the yystype.h file, adds a main function and adds the yyerrror and yywrap functions. Type the following commands to see the differences between the starting templates and original book version.
diff first.l book/
diff second.y book/

[Optional] You can change directory into the book directory and run the book.sh file to compile, link and test the original version of the software (math0.txt).
cd book/
./book.sh
cd ..

Look at the test files (math*.txt) and examine the input strings. The first file uses expressions with strings that should work with the original software. The others create syntax errors.

Look at the calc.c file. This is the main program and calls into yyparse. This version is not the same as used before in the previous assignments. Only the +yydebug flag works for debugging.

Look at the yystype.h file. Notice the book defines YYSTYPE as a double at the top of the yacc file. This version defines YYSTPE as a double inside the yystype.h file.

Look at the first.l file and notice the include files at the top of the file. The yywrap function is included at the bottom of the file.

Look at the second.y file and notice the include files at the top of the file. The yyerror function is included at the bottom of the file.

Build the calculator compiler (interpreter) calc by using the commands: **make clean** and **make**

To test this version of the calculator program from the command line type:
./calc and enter a math expression: **13e1+1e2**
You should get the result (230) printed to the standard output.
Notice on a syntax error the software does not exit: **2+3**
To exit, type on the command line: **$**

To test with yydebug statements type:
./calc +yydebug and enter a math expression as before.

To test with the first example file type:
./calc < math0.txt
Your output should be:

```
for caseid start time: Mon Sep 30 09:54:13 2013
SIZEOF int: 4 INTEGER: 8 INTEGER INTEGER: 8 float: 4 FLOAT: 8, INTEGER FLOAT:
16
Enter calculator expression and $ to exit
230
6000
-170
0.15
170
-230
6000
-6000
6000.15
```

On $ or EOF the program exits. You are now ready to solve the laboratory assignment.

Part 3: Laboratory Assignment
You will edit the current version that supports double numbers with math operations to include long long and long double values. You use type in the structure to know which value is encoded.

Edit calc.c and change all caseid to your Case ID and save the calc.c file.

Edit yystype.h and change the YYSTYPE to a structure definition to support long long and long double.

```
/*
 *     define yystype
 */
#define YYSTYPE struct yycalc
YYSTYPE
{
       int          type;        // holds INTEGER or FLOAT from %token
       long long    lvalue;      // holds INTEGER value
       long double  dvalue;      // holds FLOAT value
};
```

Save the yystype.h file.

Edit the lex.l file and remove the number definition. Then add to the top the lex macro definitions from the ansi_c compiler (hw03 scan.l). Insert this code before the first %{ as shown below.

```
D                   [0-9]
L                   [a-zA-Z_]
H                   [a-fA-F0-9]
E                   [Ee][+-]?{D}+
FS                  (f|F|l|L)
IS                  (u|U|l|L)*
```

Then between the %%s remove the lex expression {number} and action for NUMBER and insert the code below. This action encodes a hexadecimal string into the lvalue of the yylval structure.

```
0[xX]{H}+{IS}? { sscanf( yytext, "%Lx", &yylval.lvalue); yylval.type = INTEGER;
return( INTEGER); }
```

Then add action code to the regular expression shown below to support octal (%Lo), decimal (%Ld) and the three floating-point (%Lf) numbers. For the integer types use INTEGER for the type and return INTEGER. Encode the floating-point numbers into the dvalue field (&yylval.dvalue) and set the type to FLOAT and return the FLOAT value.

```
0{D}+{IS}?
{D}+{IS}?
{D}+{E}{FS}?
{D}*"."{D}+({E})?{FS}?
{D}+"."{D}*({E})?{FS}?
```

Save the first.l file.

Edit the second.y file and remove the %token NUMBER and replace it with two new tokens.

```
%token INTEGER
%token FLOAT
```

Change the precedence operators to include the logical bitwise operators.

```
%left '|'
%left '^'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%right UMINUS '~' /* supplies precedence for unary minus */
```

Change the number production with NUMBER to now be INTEGER or FLOAT.

```
number : INTEGER
       | FLOAT
       ;
```

Then change all the production actions to handle the new operand data types. For example replace the old line to print the output result with the code shown below. Notice you need to switch on the operand type to determine the format string to use in the printf statement.

```
lines  : lines expr '\n'
       {
              switch( $2.type)
              {
              case INTEGER:
                     printf("%Ld\n", $2.lvalue);
                     break;
              case FLOAT:
                     printf("%Lg\n", $2.dvalue);
                     break;
              }
       }
       | lines error '\n' { yyerror(" reenter previous line: "); yyerrok; }
       | /* empty */
       ;
```

Remove and replace all the actions for the math operators and add the bitwise operators to handle the INTEGER and FLOAT operand types. One method is shown below. Notice the action is to switch on both of the operand types and type cast the values before the operations. Save the second.y file.

```
       | expr '-' expr
       {
              switch( $1.type)
              {
              case INTEGER:
                     switch( $3.type)
                     {
                     case INTEGER:
                            $$.type = INTEGER;
                            $$.lvalue = $1.lvalue - $3.lvalue;
                            break;
                     case FLOAT:
                            $$.type = FLOAT;
                            $$.dvalue = (long double)$1.lvalue - $3.dvalue;
                            break;
                     }
                     break;
              case FLOAT:
                     switch( $3.type)
                     {
                     case INTEGER:
                            $$.type = FLOAT;
                            $$.dvalue = $1.dvalue - (long double)$3.lvalue;
                            break;
                     case FLOAT:
                            $$.type = FLOAT;
                            $$.dvalue = $1.dvalue - $3.dvalue;
                            break;
                     }
                     break;
              }
       }
```

Test your calculator program using each of the test files by typing the commands:

make clean
make
./calc < ../hw06/math0.txt
./calc < ../hw06/math1.txt
./calc < ../hw06/math2.txt
./calc < ../hw06/math3.txt
./calc < ../hw06/math4.txt
./calc < ../hw06/math5.txt
./calc < ../hw06/math6.txt
./calc < ../hw06/math7.txt
./calc < ../hw06/math8.txt
./calc < ../hw06/math9.txt

Part 4: Extra Credit (5 points)
Edit the files again and change the code to remember the format of the constants (decimal, octal, hexadecimal, floating-point). Decide on a method to determine the output results when the inputs are not using the same format (0x0ff & 1.3 -> 0x01).

Generate your output the same as in Part 5 and upload **ONLY** your first.l file to blackboard in the assignment area. Submit it with the comments **hw06** with your **CaseID** on the title line. The extra credit will be graded on-line and your score will be added to your assignment score. When you generate your laboratory output file in Part 5, show the improved type format handling. Mark your output file with **EXTRA CREDIT** and turn in with your assignment.

Part 5: Laboratory Output Generation
When all your lab assignments have been completed execute the homework script file "./hw06_test.sh". To redirect the standard error (stderr) and standard output (stdout) to a file use the following command.
./hw06_test.sh &> hw06_test.txt

Print out the **hw06_test.txt** and put your name, assignment number, date on it and turn it in with your homework exercises.

Your final directory structure for the calc compiler should be as below (using your Case ID):
EECS337/caseid/hw06/Makefile
EECS337/caseid/hw06/calc
EECS337/caseid/hw06/calc.c
EECS337/caseid/hw06/first.l
EECS337/caseid/hw06/hw06_caseid.tar
EECS337/caseid/hw06/hw06_test.sh
EECS337/caseid/hw06/hw06_test.txt
EECS337/caseid/hw06/math0.txt
EECS337/caseid/hw06/math1.txt
EECS337/caseid/hw06/math2.txt
EECS337/caseid/hw06/math3.txt
EECS337/caseid/hw06/math4.txt
EECS337/caseid/hw06/math5.txt
EECS337/caseid/hw06/math6.txt
EECS337/caseid/hw06/math7.txt
EECS337/caseid/hw06/math8.txt
EECS337/caseid/hw06/math9.txt
EECS337/caseid/hw06/second.y
EECS337/caseid/hw06/yystype.h