CS 367 001/003 Project #1:
Floating Point Representation
Due: Saturday, October 1st, at 11:59pm

This is to be an individual effort. No partners.
No late work allowed after 48 hours; each day late automatically uses up one of your tokens (or when you're out of tokens, causes a penalty – see the syllabus)

In class, we talked about the IEEE standard for floating point representation and did examples using different sizes for exponent and fraction fields so that you could learn how to do the conversions. For this assignment, **you are going to write code to do this, allowing you to store these smaller floating point numbers in a 32-bit integer**.

INPUT: For this assignment, you will read in a 'program' and call your functions to implement these programs. An example of one of these programs is:

```
x = 18.113
print x
y = 4.5
a = x + y
print a
z = x * y
print z
```

OUTPUT: The output will be the current values of the given variables at the print statements. For the above program, output would be:

x = 18.000000
a = 22.500000
z = 81.000000 *(now corrected for our specific bit-widths!)*

Some of this task is already done for you. I will provide a program that reads in the given programs, saves the variable values and calls the functions (described next) that you will be implementing.

You are going to implement a 13 bit floating point representation, where 5 bits are for the exponent and 7 are for the fraction. Using bit level operators, you will write functions (shown below) to help implement the program statements:

- **Assignment statement** (variable = value) – calls your function **`computeFP()`**, which converts from a C float value to our mini-float representation.

```
int  computeFP(float val) { }
// input: float value to be represented
// output: integer version in our representation
```

  - Given the number of bits, the rounding you will have to do for this representation is pretty substantial. For this assignment, we are always going to take the easy way and truncate the fraction (i.e. round down). For example, the closest representable value for 18.113 (rounding down) is 18.0, as can be seen in the program output.

- **Print statement** (print variable) – uses your **`getFP()`** function to convert from our mini-float representation to a regular C float value, and formats/prints it out nicely.

```
float getFP(int val) { }
// Using the defined representation, compute and
// return the floating point value
```

- **Add statement** – for this statement, you are going to take two values in our representation and use the same technique as described in class/comments to add these values and return the result in our representation.

```
int addVals(int source1, int source2) {}
```

- **Multiply statement** – for this statement, you are going to take two values in our representation and use the same technique as described in class/comments to multiply these values and return the result in our representation.

```
int multVals(int source1, int source2) {}
```

Assumptions

To make your life a little easier, we are going to make the following assumptions:

- No negative numbers. The sign bit can be ignored.
- No denormalized (or special) numbers. If the given number is too small to be represented as a normalized number, you can return 0. Same thing with numbers that are too large.

Getting Started

First, get the starting code (P1_handout.tar) from the same place you got this document. Once you un-tar the handout on zeus (using "tar xvf P1_handout.tar"), you will have the following files:

- fp_functs.c – **This is the file you will be modifying (and submitting).** There are stubs in this file for the functions that will be called by the rest of the framework. Feel free to define more functions if you like but put all of your code in this file!

- Makefile – to build the assignment (and clean up).

- README – read it.

- fp_program.c – This is the main program for the assignment. You should not change it. It implements a recursive descent parser to read in the program files, determine what each line is supposed to do, and call your functions to convert, add and multiply.

- output_all_values.c – This is a program I wrote to make debugging easier for me. It prints out all legal values in our representation! This will help you determine what values you should be seeing. For example, in the above program I assign 18.113 to x. This number is not in the output for this program. The closest smaller number is 18.0 – when I see this output, I know that value is being rounded and stored correctly.

- sampleprogram1, sampleprogram2 – some sample input files. Note that the comments give information about the expected outputs.

- fpParse.h , fp.h, and fp.l – You can ignore these files - They are the Lex specification which tokenizes input and sends it to the recursive descent parser in the main program.

Implementation Notes

- Program Files – The accepted syntax is very simplistic and it should be easy to write your own programs to test your code (which I strongly encourage). Variable names are single lower case letters. Comments start with the character '#" and go to the end of the line. There are 4 different statement types:
  - **`print x`** - where x is a variable.
  - **`x =`** *value* - for some floating point value. This statement has the obvious meaning.
  - **`x = y + z`** - for any legal variable names
  - **`x = y * z`** - for any legal variable names

Submitting & Grading

Submit this assignment electronically on blackboard. In addition, bring a hard copy of this file to the first class after the due date. Note that the file that gets submitted is **`fp_functs.c`**

Your grade will be determined as follows:

- 30 points - code & comments. Be sure to document your design clearly in your code comments. This score will be based on reading your source code (you will lose points if you do not turn in a hardcopy).
- 70 points – correctness. We will be building your code using the **fp_functs.c** code you submit along with our code (which will be the framework with some extensions to make our grading easier). If you program does not compile, we cannot grade it. If your program compiles but does not run, we cannot grade it. I will give partial credit for incomplete submissions. You will not get credit for a particular part of the assignment (multiplication for example), if you do not use the required techniques, even if your program performs correctly on the test cases for this part.