CS 367 Project 2 - Fall 2022:

KIFP Software Floating-Point Library

Due: Friday, October 7, 11:59pm

This is to be an individual effort. No partners. No Internet code/collaboration.
Protect your code from anyone accessing it. Do not post solutions on public repositories.

No late work allowed after 48 hours; each day late automatically uses up one of your tokens.

Core Topics: Floating Point, Bitwise Operators, C Programming Design (Helper Functions)

1. Project Overview

The Zeus User Operations Notary (**ZUON**) Programming Language is a python-like interpreted programming language designed to run on an **embedded system** without any hardware floating-point support. Many such embedded systems typically do not have support for floats in hardware. You won't have to worry about any of this, of course, but it means that you won't be able to use any normal floats or doubles. **You may not use any C float or double types.**

Your job is to implement KIFP, which is a custom 9-bit floating-point library.

You will be implementing functions to create the **KIFP** 9-bit floating point library that ZUON will be using. You will be completing six functions in src/**kifp.c** for the API, however, it is **highly** recommended that you write a large number of helper functions.

In addition to encoding and decoding KIFP values, you will also do some native arithmetic in this format, using the techniques covered in class, operating on **kifp_t** values.

2. The ZUON Programming Language

The good news is that the ZUON programming language is already written; all you have to do is finish the API implementation for the six **KIFP** functions in **kifp.c** that ZUON will use. This language is normally programmed interactively from the command line without any inputs, in which case it will let you type in your operations, one per line, for it to execute.

ZUON supports any variable that starts with a letter: (eg. **foo**, **B**, or **o_b_1**)
ZUON has four different arithmetic operators: **=** **+** **-** *****
ZUON has a negation operator: **-**
(eg. When you enter -2, we convert 2 first to KIFP, then we do the negation!)
ZUON has two constants: **inf** **nan**
ZUON has two functions: **print()** and **display()**
ZUON has one command: **quit** (**exit** also works)
ZUON does single-line comments: **#** (eg. **# this is a comment**)

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

You may also use scripts, like how Python scripts are run. The ZUON scripts use the **.zuon** extension and run the commands that will ultimately call your functions.

Here is a summary of how ZUON calls your functions:

Any number entered will call your **toKifp** function to convert it into a KIFP Value (kifp_t)
ZUON Functions:

print()	Calls your **toNumber** to convert a kifp_t value to a Number
display()	Debug Function that will display any kifp_t value in Binary

Arithmetic Operators:

+	Calls your **addKifp** to add two kifp_t values and return a kifp_t result.
-	Calls your **subKifp** to subtract two kifp_t values and return a kifp_t result.
*****	Calls your **mulKifp** to multiply two kifp_t values and return a kifp_t result.

Negation Operator:

-	Calls your **negateKifp** to negate a kifp_t values and return a kifp_t result.

3. ZUON Programming Language Output

ZUON outputs directly to the screen with the results of the operations that are given.

ZUON Programming Language Scripts

If you run a script, the output will be given all at once.

Here is an example **ZUON** script in (scripts/sample.zuon) with one statement per line:

```
# Sample Script
first = -0.45
print(first)
second = 4.5
print(second)
sum = first + second
print(sum)
difference = first - second
print(difference)
product = first * second
print(product)
complex = 3 + second - 1.25 * -2
print(complex)
display(complex)
quit
```

This script inputs 14 commands into the ZUON Programming language.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

Here is the output for the provided sample script.

```
kandrea@zeus-2:handout$ ./zuon < scripts/sample.zuon
Welcome to the Zeus User Operations Notary (ZUON) programming language.
(ﾉ°□°)ﾉ︵ ┻━┻:  $ # Sample Script
(ﾉ°□°)ﾉ︵ ┻━┻:  $ first = -0.45
(ﾉ°□°)ﾉ︵ ┻━┻:  $ print(first)
first = -0.4453125
(ﾉ°□°)ﾉ︵ ┻━┻:  $ second = 4.5
(ﾉ°□°)ﾉ︵ ┻━┻:  $ print(second)
second = 4.5
(ﾉ°□°)ﾉ︵ ┻━┻:  $ sum = first + second
(ﾉ°□°)ﾉ︵ ┻━┻:  $ print(sum)
sum = 4.0
(ﾉ°□°)ﾉ︵ ┻━┻:  $ difference = first - second
(ﾉ°□°)ﾉ︵ ┻━┻:  $ print(difference)
difference = -4.875
(ﾉ°□°)ﾉ︵ ┻━┻:  $ product = first * second
(ﾉ°□°)ﾉ︵ ┻━┻:  $ print(product)
product = -2.0
(ﾉ°□°)ﾉ︵ ┻━┻:  $ complex = 3 + second - 1.25 * -2
(ﾉ°□°)ﾉ︵ ┻━┻:  $ print(complex)
complex = 10.0
(ﾉ°□°)ﾉ︵ ┻━┻:  $ display(complex)
KIFP Value in Binary = : 011001000 (0x0c8)
(ﾉ°□°)ﾉ︵ ┻━┻:  $ exit
Thanks for all the Fish!

Have a nice day!
```

ZUON Programming Language Interpreter

You can also run **./zuon** directly from the command line without a script, just like with a Python interpreter. You can type each of the same lines in by hand and you'll get the same output.

```
kandrea@zeus-1:handout$ ./zuon
Welcome to the Zeus User Operations Notary (ZUON) programmable calculator.
(ﾉ°□°)ﾉ︵ ┻━┻:  $ foo = 0.25
(ﾉ°□°)ﾉ︵ ┻━┻:  $ bar = -1.5
(ﾉ°□°)ﾉ︵ ┻━┻:  $ print(foo * 2 + bar)
Value = -1.0
(ﾉ°□°)ﾉ︵ ┻━┻:  $ print(-inf)
Value = -Infinity
(ﾉ°□°)ﾉ︵ ┻━┻:  $ display(inf)
KIFP Value in Binary = : 011100000 (0x0e0)
(ﾉ°□°)ﾉ︵ ┻━┻:  $ print(0 * -inf)
Value = NaN
(ﾉ°□°)ﾉ︵ ┻━┻:  $ quit
Thanks for all the Fish!

Have a nice day!
```


Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

4. Specification for Project 2

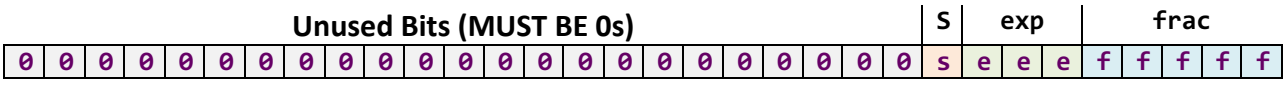
We've already written the ZUON Programming language for you and provided you with the stubs (empty functions) for the six functions you will be writing inside of src/**kifp.c**

Complete this code, along with any number of helper functions that you would like to use, in order to implement these three functions. In this project, you will be working with our custom **kifp_t** type variables. These custom types are **32-bit signed ints** in memory. Within these 32-bits, you will be encoding our custom 9-bits floating point value.

Since a **kifp_t** type is just a standard **int**, you can do operations on it just like you normally would with a signed int. (eg. shifting, masking, and other bitwise ops). Ultimately, you will be getting the **S**, **exp**, and **frac** information and storing them within your **kifp_t** value, just like we've been doing in class.

KIFP Representation (kifp_t) Values

The **kifp_t** values are encoded using the following format within a signed 32-bit int:



This is the 9-bit Representation for **kifp_t** values:

1 bit for sign (s), 3 bits for exponent (exp) and 5 bits for fraction (frac).

KIFP is a slightly simplified version of floating-point encoding. We can support **Normalized**, **Denormalized**, and Special (NaN or ∞) encodings. ZUON does support rounding, but only Round-To-Zero (**Truncation**).

Rules describing the outcomes of arithmetic operations will be discussed later.

Input Values to KIFP

Since this is a program for a computer that does not have formal floating-point support, you may notice that there are no **float** or **double** types anywhere in the code for this program. In fact, you will not be allowed to use any **float** or **double** types in your solution either.

Because of this, the inputs to the functions you will be writing will be **int** types inside of a struct. These values will give you all of the information that you need to use in order to implement your custom floating-point type (**kifp_t**) values.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

Number Struct Definition (inc/common_structs.h):

```
typedef struct number_struct {
  char original[255];   // Reference String for ZUON, Ignore this Field
  int is_negative;      // 1 if Negative, 0 if Positive
  int is_infinity;      // 1 if Infinity, 0 Otherwise
  int is_nan;           // 1 if NaN, 0 Otherwise
  int whole;            // 32-bit Whole Portion of Number in Binary
  //   (eg. For 3.25, whole = 0x3)
  int fraction;         // 32-bit Fraction Portion in Binary
  //   (eg. For 3.25, frac  = 0x40000000, which represents .01000000000000…)
  int precision;        // Used Internally by ZUON, Ignore this Field
} Number_t;
```

Your primary two functions (**toKifp** and **toNumber**) will have this struct passed into them by reference. Do NOT free those when you are done with them. (They are controlled by ZUON).

Example: Let's say someone enters the value **3.0625** in ZUON.
ZUON will pass a Number_t struct to your **toKifp** function with the following members set:

- **original** = "3.0625" ***(Ignore this Field, this is unused for your KIFP)**
- is_negative = 0
- is_infinity = 0
- is_nan = 0
- whole = 3
- fraction = 0x10000000
  - 3.0625 is 11.0001 in Binary.
  - The fraction field is a 32-bit int that has the Binary value of the fraction.
  - So, .0001 would be .0001 0000 0000 0000 0000 0000 0000 0000 as 32-bit value.
  - fraction gives this to you in Binary as a 32-bit int.
    - 0x10000000 is 0001 0000 0000 0000 0000 0000 0000 0000 as 32-bits.
  - So, fraction gives you the fraction component AS a whole 32-bit value.

Entering Negative Values

One note on negatives is that when you type in a negative value, ZUON will first convert the number itself using **toKifp** and then will call your **negateKifp** function to negate it.

This is because ZUON is a real programming language, so when you enter the following:

```
-1.23
```

ZUON parses it as this, converting 1.23 to a kifp_t first, and then doing the negation operator:

```
-(1.23)
```

Note that negative numbers can be part of other expressions that will call **toKifp**, so pay special attention to the **is_negative** flag and make sure to set your S bit accordingly.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

Function Descriptions

src/**kifp.c** has been given to you as your starting file. This contains a stub for all of the six required functions. You are strongly encouraged to create helper functions, constants, etc. in your design. They too must all be kept within **kifp.c** as this is the only file you will be submitting.

You are not allowed to use any **float** or **double** types in this project.

Write the code for these functions, **using bitwise operators** for encoding/decoding.

KIFP Function: `kifp_t toKifp(Number_t *number)`

When ZUON gets any number, `(example: apple = 1.25)` it will call this function.

`toKifp` will take a Number Struct (with its **whole** and **fraction** parts) and encode the data into our custom 9-bit representation and return that value.

kifp_t is a typedef for a signed 32-bit **int** in C

Once you have encoded the value into this **kifp_t**, you will be returning it.

Example: The val **1.25** has Sign = 0, exp = 011, frac = 01000

The 9-bit encodings should be: `0 011 01000`
The full kifp_t (32-bit) value will be: `0000 0000 0000 0000 0000 0000 0110 1000`
(Hex: `0x068`)

So, how do you get these fields from the value?
You are not allowed to use **float** or **double** when working within your function, but you can do the same operations we did in class. (Hint: Think of Binary Scientific Notation)

Think carefully in your design about how you want to shift the whole and fraction integers, and how you will need to move values between them, in order to get the right format for Binary Scientific Notation.

Remember also that you will need to have something to track your E component and that for each shift you do, you will need to adjust that E.

The key idea is to get your value into the right range while adjusting E. This will give you the ability to determine the S, E, and M components first.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

Example for 3.25:

- number->whole is 0x3
  - This is 0000 0000 0000 0000 0000 0000 0000 00**11** in Binary
- number->fraction is 0x40000000 *(the bits on the right of the binary point)*
  - This is **01**00 0000 0000 0000 0000 0000 0000 0000 in Binary

Together, they represent 11.01 in Binary

To get this in Binary Scientific Notation for Normalized Encoding, you can shift!

- Be careful here because these are two 32-bit integers!
- Starting point:
  - number->whole is 3, which is 0x00000003
    - This is 0000 0000 0000 0000 0000 0000 0000 001**1** in Binary
  - number->fraction is 0x40000000
    - This is 0100 0000 0000 0000 0000 0000 0000 0000 in Binary
  - These represent: 1**1**.01
- Goal after shifting right once:
  - number->whole is 1, which is 0x00000001
    - This is 0000 0000 0000 0000 0000 0000 0000 0001 in Binary
  - number->fraction is 0xA0000000
    - This is **1**010 0000 0000 0000 0000 0000 0000 0000
  - These represent: 1.**1**01

Rounding:

Note that frac is big (5-bits), however it's not big enough to store all possible values, so we will need to do rounding. This will be done with a simpler technique that we discussed in class. For this project, we'll just be doing **truncation**, which means we discard any bits that don't fit. It's also the same as **round-to-zero** rounding.

As an example, **kifp** can represent **14.0** and **14.25**, but nothing in between. When **14.24** is entered into ZUON, it will round-to-zero (down) to **14.0**.

We can determine what things **should** round to by looking at the output of a provided helper program, **ref_all_values**. This program prints out every possible value that can be represented in kifp, along with the Mantissa and binary representation.

```
...
M = 1.750000 (b1.11000), val=14.000000000000000000000000000000000000000 [0x00d8]
M = 1.781250 (b1.11001), val=14.250000000000000000000000000000000000000 [0x00d9]
M = 1.812500 (b1.11010), val=14.500000000000000000000000000000000000000 [0x00da]
...
```

So, we can see that **14.0** and **14.25** are both representable, but **14.24** is not. So, since we're rounding by **truncation**, we'll always round down to the next representable value.

Return your kifp_t value if everything worked properly, or -1 if you have any errors.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

Special Rules:

- **For Underflows** (eg. exp would be ≤ 0 with $1 \leq M < 2$ in the Normalized Range):
  - Encode as Denormalized, as we covered in class.
- **Negatives**
  - Note that you can get a negative value on input and it must be handled.
- **Special Values**
  - Note that you may get either Infinity (**is_infinity**) or Nan (**is_nan**) passed in.
    - These need to be handled according to the standard encoding.
  - If you get **both** of these flags, treat the number as NaN.
- **For Overflows** (eg. exp is too large for Normalized):
  - Encode the **kifp_t** variable as the special value $\infty \; or \; -\infty$ as appropriate.
  - Specials are as we covered with the exp set to all 1s and frac set accordingly.

KIFP Function:	`int toNumber(Number_t *num, kifp_t value)`

When ZUON gets the **print** function, example: `print(foo)`, it will call this function.

`toNumber` will convert our 9-bit representation (value) into the S, M, and E and the put the resulting components into a Number_t struct that's passed in by reference.

Do NOT free or malloc on num, this is managed by ZUON and is already allocated for you!

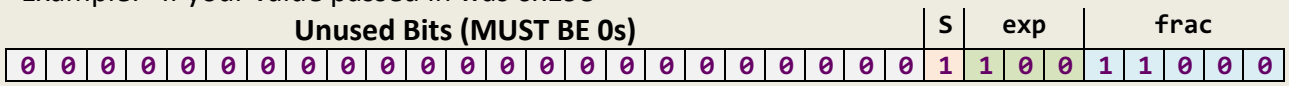
Extract the **S**, **exp**, and **frac** portions of the **`kifp_t`** type and then convert them back into normal number form (just the whole and fraction values without any multipliers) these fields of the Number_t struct. **num** is guaranteed that all values inside were initialized to 0s.

Fill in these Fields of **num**:

- **is_negative**
  - Set this to 1 if the value is negative, or 0 if positive.
- **is_infinity**
  - Set this to 1 is the value represents Infinity, or 0 otherwise.
- **is_nan**
  - Set this to 1 if the value represents NaN, or 0 otherwise.
- **whole**
  - Set this to the whole number of your converted value.
- **fraction**
  - Set this to the binary representation of the bits that would be on the right of the binary point in the converted value.
- **precision** – Unused, Leave As-Is
- **original** – Unused, Leave As-Is

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

Example: If your value passed in was 0x198



This kifp_t value encodes the number $-1^1 * 1.11 * 2^1$, which has a value of -11.1 in Binary. (-3.5)

Your toNumber function must take this kifp_t value and convert it back to the original number format, so that you would set your Number_t struct with these values:

- **is_negative = 1**
- **is_infinity = 0**
- **is_nan = 0**
- **whole = 3**
  - **3** is 0000 0000 0000 0000 0000 0000 0000 0011 in binary as a 32-bit int.
- **fraction = 0x80000000**
  - **0x80000000** is 1000 0000 0000 0000 0000 0000 0000 0000 in binary as 32-bit int.
  - This represents the bits on the RIGHT of the binary point.

You will not return num, since it's passed in by reference, so when you set the values and return, ZUON already has access to it.

Return 0 if everything worked properly, or -1 if you have any errors.

KIFP Function: `kifp_t mulKifp(kifp_t val1, kifp_t val2)`

This will multiply two kifp_t values and then return the result as a properly formatted kifp_t.

Multiplication Examples in ZUON:	**foo * bar** **bar * -3.5**

Extract the **S**, **exp**, and **frac** portions of each one of the two value arguments, convert them into S, E, and M, then multiply them together using the technique covered in class:

val1:	S1, M1, E1	
val2:	S2, M2, E2	
product:	S, M, E	
		S = S1 ^ S2
		M = M1 * M2
		E = E1 + E2

One trick you may use if you like, it may be helpful, since your M will be in terms of **whole** and **fraction** parts, would be to shift M to make each value only whole numbers first and then just multiply them as integers, at which point you can take the result and encode that back into a proper **kifp_t** value. Just remember to properly manage your E components during shifting!

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

Once you have the S, M, and E of the product, then encode them back into the **kifp_t** format and return that result or return -1 if there were any Errors.

Special Rules:

- **Rounding:**
  - The result may not fit evenly into your format anymore! Make sure to round (**truncation**) to fit into the frac field.
- **For Underflows** (eg. exp would be ≤ 0 with $1 \leq M < 2$ in the Normalized Range):
  - Encode as Denormalized.
- **For Overflows** (eg. exp is too large for Normalized):
  - Encode the **kifp_t** variable as the special value $\infty \; or \; -\infty$ as needed.
- **Sign Considerations:**
  - Always follow the multiplication sign rules.
- **Arithmetic Special Cases:** See the next section on **Arithmetic Rules**

You are not allowed to use double or float data types at any point in these functions.

KIFP Function: `kifp_t addKifp(kifp_t val1, kifp_t val2)`

This will add two kifp_t values and then return the result as a properly formatted kifp_t.

Addition Examples in ZUON:	**foo + bar** **2.5 + 3.5**

Extract the **S**, **exp**, and **frac** portions of each one of the two value arguments, convert them into S, E, and M, then add them together using the technique covered in class:

value1: S1, M1, E1
value2: S2, M2, E2
result: S, M, E

Align the Mantissas:

You need both Es to be the same, so pick one of your values and adjust it. Once both E1 and E2 are equal, you can now add your Ms to get the resulting M of the sum. The E of the result will then be E1.

For the Sign, you will need to determine what it should be.

Example, 5.0 + -3.0 will be positive, while -5.0 + 3.0 will be negative.

Once you have S, M, and E of result, encode them into **kifp_t** format and return the result.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

One trick you may use if you like, it may be helpful, since your M will be in terms of **whole** and **fraction** parts, would be to shift M to make each value only whole numbers first and then just add them as integers, at which point you can take the result and encode that back into a proper **kifp_t** value. Just remember to properly manage your E components during shifting!

Once you have the S, M, and E of the sum, then encode them back into the **kifp_t** format and return that result or return -1 if there were any Errors.

Special Rules:

- **Rounding:**
  - The result may not fit evenly into your format anymore! Make sure to round (**truncate**) to fit into the frac field.
- **For Underflows** (eg. exp would be ≤ 0 with $1 \leq M < 2$ in the Normalized Range):
  - Encode as Denormalized.
- **For Overflows** (eg. exp is too large for Normalized):
  - Encode the **kifp_t** variable as the special value $\infty \; or \; -\infty$ as needed.
- **Arithmetic Special Cases:** See the next section on **Arithmetic Rules**

You are not allowed to use double or float data types at any point in these functions.

KIFP Function: `kifp_t subKifp(kifp_t val1, kifp_t val2)`

This will perform the following subtraction: **val1 – val2**

Subtraction Examples in ZUON:	**foo - bar**
	2.5 - 3.5

Extract the **S**, **exp**, and **frac** portions of each one of the two value arguments, convert them into S, E, and M, then add them together using the technique covered in class:

value1: S1, M1, E1
value2: S2, M2, E2
result: S, M, E

Align the Mantissas:

You need both Es to be the same, so pick one of your values and adjust it.
Once both E1 and E2 are equal, you can now subtract for M1 – M2 to get the resulting M of the difference. The E of the result will then be equal to E1 and E2.

For the Sign, you will need to determine what it should be.
Example, 5.0 + -3.0 will be positive, while -5.0 + 3.0 will be negative.

Once you have S, M, and E of result, encode them into **kifp_t** format and return the result.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

One trick you may use if you like, it may be helpful to shift M to be whole numbers first and then subtract them as integers, at which point you can take the result and encode that back into a kifp_t value. Just remember to properly manage your E components!

Once you have the S, M, and E of the sum, then encode them back into the **kifp_t** format and return that result or return -1 if there were any Errors.

Special Rules:

- **Rounding:**
  - The result may not fit evenly into your format anymore! Make sure to round (**truncate**) to fit into the frac field.
- **For Underflows** (eg. exp would be ≤ 0 with $1 \leq M < 2$ in the Normalized Range):
  - Encode as Denormalized.
- **For Overflows** (eg. exp is too large for Normalized):
  - Encode the **kifp_t** variable as the special value $\infty \; or \; -\infty$ as needed.
- **Arithmetic Special Cases:** See the next section on **Arithmetic Rules**

You are not allowed to use double or float data types at any point in these functions.

KIFP Function: `kifp_t negateKifp(kifp_t value)`

This will negate a kifp_t value.

Negation Examples in ZUON:	**-bar** **-3.5**

Change the sign of the input value and then output that result in **kifp_t** format.

Return the **kifp_t** result or return -1 if there were any Errors.

You are not allowed to use double or float data types at any point in these functions.

The remaining ZUON Commands are already written for you.

ZUON Function: display(variable)

display Example in ZUON:	**display(bar)**

This command shows the binary representation of your **kifp_t** values.
It's great for debugging! It's also written for you, so no work is needed.

Example:

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

```
(ﾉ°□°)ﾉ ┻━┻:  $ bar = 1.25
(ﾉ°□°)ﾉ ┻━┻:  $ display(bar)
KIFP Value in Binary = : 001101000 (0x068)
```

This represents a kifp_t value with these bits:

0 011 01000

S = 0, exp = 011, frac = 01000

This shows the bits of the variable you set with your earlier functions, as well as a quick representation in hex (0x068) to help you compare easier with the ref_all_values program:

M = 1.250000 (b1.01000), val=1.250000000000000000000000000000000000 [0x0068]

ZUON Function: print(variable)

display Example in ZUON: print(bar)

This command shows you the value of a variable or an expression. This will call your **toNumber** function whenever it's called.

Example:

```
(ﾉ°□°)ﾉ ┻━┻:  $ bar = 1.25
(ﾉ°□°)ﾉ ┻━┻:  $ print(bar)
bar = 1.25
(ﾉ°□°)ﾉ ┻━┻:  $ print(3.0 - 1.5)
Value = 1.5
```

ZUON Operator: quit

exit Examples in ZUON: quit
exit

You can use either **exit** or **quit** to quit the program. Easier than leaving Python!

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

5. Notes on ZUON

ZUON is a programming language, just like Python is, and as such, it does process your expressions like a programming language would.

As an example, ZUON can do this:

```
Welcome to the Zeus User Operations Notary (ZUON) programming language.
(ﾉ°□°)ﾉ ︵ ┻━━┻: $ foo = 3.5
(ﾉ°□°)ﾉ ︵ ┻━━┻: $ print(foo + 5.6 + bar = 2.5)
Value = 11.5
(ﾉ°□°)ﾉ ︵ ┻━━┻: $ print(bar)
bar = 2.5
(ﾉ°□°)ﾉ ︵ ┻━━┻: $ print(-1)
Value = -1.0
```

Every expression will return the value, so you can combine operations like you would with Python, for instance. Each operation is performed in order of precedence and will result in a KIFP value.

Since every expression results in a KIFP value, by typing in 3.5 ZUON will convert it using **toKifp** while processing that statement.

The last command in the above example is an interesting one. print(-1)

```
(ﾉ°□°)ﾉ ︵ ┻━━┻: $ print(-1)
```

This is interesting because ZUON first takes the 1 as a number and calls your **toKifp**() function on the value 1.0. The result of that that you return will then be operated on by the **–** (negation) operator. In this case, putting a – in front of a number will make it negative by first having the number converted to a KIFP value with **toKifp**(), and THEN will call **negateKifp**() on the result!

You also have two special values you can use in ZUON:

- **inf** Infinity
- **nan** NaN

Make sure your **toKifp** can handle inputs of infinity (number->is_infinity == 1) and NaN (number->is_nan == 1) on inputs.

In the unexpected case of both being set, treat the number as NaN.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

```
Welcome to the Zeus User Operations Notary (ZUON) programming language.
(ﾉ°□°)ﾉ ⌒┻━┻: $ a = inf
(ﾉ°□°)ﾉ ⌒┻━┻: $ print(a)
a = Infinity
(ﾉ°□°)ﾉ ⌒┻━┻: $ a = nan
(ﾉ°□°)ﾉ ⌒┻━┻: $ print(a)
a = NaN
(ﾉ°□°)ﾉ ⌒┻━┻: $ a = -inf
(ﾉ°□°)ﾉ ⌒┻━┻: $ print(a)
a = -Infinity
```

6. Special Arithmetic Rules

Use these rules for special cases when doing arithmetic:
(X represents any Real number that is not 0, NaN, or ∞)

1. **Addition Special Rules** *(Arguments can be in any order of mathematical equivalence)*
   - $\infty + \infty = \infty$
   - $\infty - \infty = NaN$
   - $-\infty - \infty = -\infty$
   - $NaN \pm Anything = NaN$
   - $X - X = 0$
   - $\infty \pm X = \infty$
   - $-\infty \pm X = -\infty$
   - $0 + 0 = 0$
   - $0 - 0 = 0$
   - $-0 - 0 = -0$
   - $-0 + 0 = 0$
   - $0 \pm \infty = \pm\infty$ $(sign\ appropriate)$
   - $0 \pm X = \pm X$ $(same\ Value\ sign)$
   - $-0 \pm X = \pm X$ $(same\ Value\ sign)$

2. **Multiplication Special Rules** *(Args can be in any order of mathematical equivalence)*
   $Follow\ multiplication\ rules\ for\ Signs\ (e.g. -\infty * -\infty = \infty\ or\ -0 * 4 = -0)$
   - $\infty * \infty = \infty$
   - $NaN * Anything = NaN$
   - $\infty * X = \infty$
   - $\infty * 0 = NaN$
   - $0 * X = 0$

Remember the sign rules for any Multiplication!
(Multiplying by ∞, -∞, 0 or -0 is handled using normal multiplication sign rules)

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

7. Project Constraints

You may Not #include <math.h> or use any math.h functions, including pow()

You may Not use any double or float types anywhere in your code.

There are Two Special Number Types: Infinity and NaN.

- These will be implemented using the standard special number pattern in your **kifp_t** floating-point representation. (Remember $\infty \; and \; -\infty$)
- There is only one NaN, regardless of sign, it's not a number.
  - Any pattern which matches a NaN is considered equivalent.
  - **Your kifp_t inputs should be able to recognize any value bit representation of NaN.**
  - Your functions may use any valid NaN representation.

Rounding is by Truncation (eg. round-to-zero)

- If your frac is too big for the field, truncate and only fill the field with the bits that fit in the field.

Negative Numbers must be handled.

- All values (including ∞) will be handled properly with negatives.
- All functions should support -0 values being passed in as arguments.

8. Getting Started

First, get the starting code (`project2_handout.tar`) from the same place you got this document. Once you un-tar the handout on zeus (using `tar xvf project2_handout.tar`), you will have the following key folders and key files:

- **Makefile** – Run **make** to build the assignment (and **make clean** to clean up).
- **src/**
  - **kifp.c** – **This is the only 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!
- **inc/**
  - **kifp.h – Do Not Modify.** This has some nice constants that you may wish to use if you like.
  - **common_structs.h – Do Not Modify.** This has the struct definitions for the project.
  - **zuon_settings.h:** You may modify this. This defines what the prompt looks like and whether or not you want to see the colors. Feel free to change as you like.
- **scripts/**
  - **sample.zuon** – This is the provided sample ZUON script. You can write your own for testing!

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

- **ref_all_values** – This is a program we wrote to make debugging easier for us. It prints out all legal values in our representation. This will help you determine what values you should be seeing.

ref_all_values Program

For each E (the E is given and the exp equivalent is given in binary), `ref_all_values` lists all possible values that can be represented (vals). For each val, you get the M in decimal and in binary for convenience.

For example, let's say we assign `4.45` to **foo**. This number is not a valid **val** in the output for this program, as shown in the snippet from the all_values output below.

```
 ...
  M = 1.093750 (b1.00011), val=4.375000000000000000000000000000000000000 [0x00a3]
  M = 1.125000 (b1.00100), val=4.500000000000000000000000000000000000000 [0x00a4]
...
```

The closest values to **4.45** are `4.375` and `4.5`. Since we're using rounding by truncation (round-to-zero), this will round down to `4.375` as seen in the following output.

```
(ﾉ°□°)ﾉ⌒ ┻━━┻:  $ foo = 4.45
(ﾉ°□°)ﾉ⌒ ┻━━┻:  $ display(foo)
KIFP Value in Binary = : 010100011 (0x0a3)
(ﾉ°□°)ﾉ⌒ ┻━━┻:  $ print(foo)
foo = 4.375
```

Of course, doing this in decimal is very hard. It's a lot easier once you're working in the code to do the rounding from the binary directly to keep the bits that fit in frac.

9. Implementation Notes

- **ZUON (.zuon)** Script Files – The accepted syntax is very simplistic and it should be easy to write your own scripts to test your code (which we strongly encourage).
  - **ZUON** only uses standard variable names (any that start with a letter).
  - **ZUON** command reference:
    - `print(x)` Prints the Number. **toNumber()**
    - `display(x)` where x is a variable to display the bit representation
    - `x = value` Performs assignment. **toKifp()**
    - `y + z` Performs addition. **addKifp()**
    - `y - z` Performs subtraction. **subKifp()**
    - `y * z` Performs multiplication. **mulKifp()**
    - `-x` Performs negation. **negateKifp()**
    - `quit` Exits the program. (exit also works)
    - `#` Single-Line Comment

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

Note that you can chain operators just like in Python too:

```
(ノ°□°)ノ︵ ┻━━┻:  $ foo = 4.45
(ノ°□°)ノ︵ ┻━━┻:  $ display(foo)
KIFP Value in Binary = : 010100011 (0x0a3)
(ノ°□°)ノ︵ ┻━━┻:  $ print(foo)
foo = 4.375
(ノ°□°)ノ︵ ┻━━┻:  $ foo = 1.0
(ノ°□°)ノ︵ ┻━━┻:  $ bar = -0.25
(ノ°□°)ノ︵ ┻━━┻:  $ result = foo + foo * -bar + 3.5 * -foo
(ノ°□°)ノ︵ ┻━━┻:  $ print(result)
result = -2.25
```

- As a programming language, we also have operator precedence, but fortunately this is pretty much as expected.
  - Parentheses are highest precedence.
  - Negation is next highest.
  - Multiplication is next highest.
  - Addition and Subtraction are next.
  - The Assignment (=) is the lowest.

```
(ノ°□°)ノ︵ ┻━━┻:  $ a = -(3+2)
(ノ°□°)ノ︵ ┻━━┻:  $ print(a)
a = -5.0
(ノ°□°)ノ︵ ┻━━┻:  $ a = -3+2
(ノ°□°)ノ︵ ┻━━┻:  $ print(a)
a = -1.0
```

- If you run **ZUON** from the command line without inputting a script file, you must end the session with either **quit** or **exit**.
- To run **ZUON** with a script file, you can pass it in with redirects:
  `./ZUON < scripts/sample.zuon`

Note on Types:

Remember that in C, all integer types are just collections of bits. So, an **int** is just 32-bits. We can interpret this as an Integer, of course, but we can also just use it as a container and work with those individual bits.

The **kifp_t** type is an int internally, but you can think of it as a 32-bit container to use. Just remember if you do any shifting, C will still perform right-shifts based on the signed/unsigned type rules.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

10. Submitting & Grading

Submit this assignment electronically on Blackboard. Note that the only file that gets submitted is **`kifp.c`**. Make sure to put your name and G# as a commented line in the beginning of your source file.

You can make multiple submissions; but we will test and grade ONLY the latest version that you submit (with the corresponding late penalty, if applicable).

Important: Make sure to submit the correct version of your file on Blackboard! Submitting the correct version late will incur a late penalty; and submitting the correct version 48 hours after the due date will not bring any credit.

Questions about the specification should be directed to the CS 367 Piazza forum.

A full Rubric is available on Blackboard for the Project as well with more details.

Your grade will be determined as follows:

- **20 points** - code & comments. Be sure to document your design clearly in your code comments. This score will be based on reading your source code.
- **80 points** – correctness. We will be building your code using the **kifp.c** code you submit along with our code.
    - If you program does not compile, **we cannot grade it**.
    - If your program compiles but does not run, **we cannot grade it.**
    - We will give partial credit for incomplete programs that build and run.
    - 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.

Valgrind Notes:

This program leaks more than if the Titanic crashed into the Poseidon in an Aquaman movie.

No Valgrind will be Run and No Leak Checks will be Made.

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved

11. Changelog

v1.0: September 15th – Initial Release

Copyright: Prof. Kevin Andrea, George Mason University, 2022. All Rights Reserved