CS 367 Project 2 – Fall 2024:
MUAN Programming Language Manual

1. The MUAN Programming Language

The MUAN programming language is already written; all you must do is finish the API implementation for the six **MicroFP** functions in **microfp.c** that MUAN 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.

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

Sample Execution:

```
kandrea@zeus-2:handout/ $ ./muan
Welcome to the Micro-Ubiquitous Accounting Notary (MUAN) programmable calculator.
¯\_(ツ)_/¯ $ print(1.0)
Value = 1.0
¯\_(ツ)_/¯ $ x = 2.5
¯\_(ツ)_/¯ $ y = -0.25
¯\_(ツ)_/¯ $ speed = 10.3
¯\_(ツ)_/¯ $ display(x)
MicroFP Value in Binary: 0 100 01000 (0x088)
¯\_(ツ)_/¯ $ print(x)
x = 2.5
¯\_(ツ)_/¯ $ print(-x)
Value = -2.5
¯\_(ツ)_/¯ $ x++
¯\_(ツ)_/¯ $ print(x)
x = 3.5
¯\_(ツ)_/¯ $ print(++x)
Value = 4.5
¯\_(ツ)_/¯ $ combo = speed - 4 + (x * 2) - y
¯\_(ツ)_/¯ $ combo -= 1.5
¯\_(ツ)_/¯ $ print(combo)
combo = 14.0
¯\_(ツ)_/¯ $ quit
SIC PARVIS MAGNA

Have a nice day!
```

MUAN Operators and Commands

MUAN supports any variable that starts with a letter:
eg. **foo**, **B**, or **o_b_1** are all valid variable names

MUAN has seven different arithmetic operators:

		Examples:
+	`Addition`	`3.14 + 1.5`
-	`Subtraction`	`3.14 - 1.5`
*****	`Multiplication`	`3.14 * 1.5`
`++`	`Pre-Increment`	`++x`
`++`	`Post-Increment`	`x++`
`--`	`Pre-Decrement`	`--x`
`--`	`Post-Decrement`	`x--`

MUAN has four different arithmetic operators:

=	`Assignment`	`foo = 3.14`
`+=`	`Addition Assignment`	`foo += 3.14`
`-=`	`Subtraction Assignment`	`foo -= 3.14`
`*=`	`Multiplication Assignment`	`foo *= 3.14`

MUAN has a negation operator:

-	`Negate`	`-x`

MUAN has two constants:

inf	Infinity (∞)	x = inf
nan	Not a Number (NaN)	x = nan

MUAN has two functions:

print(X) Prints X, where X is a variable, expression, or number.
Example:

```
print(foo)
foo = -4.25
```

display(X) Prints the MicroFP's Bits (where X is a variable, expression, or number)
Example:

```
display(1.5)
MicroFP Value in Binary: 0 011 10000 (0x070)
```

MUAN has three commands:

help Prints basic help information.
exit Exits the program.
quit Exits the program.

MUAN does single-line comments, starting with a #:
# this is a comment

MUAN Calls to MicroFP Functions

Here is a summary of how MUAN calls the six functions you are writing for this project:

Any number you enter will call **toMicroFP()** function to convert it into a MicroFP Value (**microfp_s**)

MUAN Functions:

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

Arithmetic Operators:

+ and **+=** Calls **addMicroFP()** to add two **microfp_s** values and return a **microfp_s**
- and **-=** Calls **subMicroFP()** to subtract two **microfp_s** values and return a **microfp_s**
***** and ***=** Calls **mulMicroFP()** to multiply two **microfp_s** values and return a **microfp_s**

Postfix and Prefix Increment/Decrement Operators:

`++` Calls **addMicroFP()** to add a **microfp_s** values with 1.0 and return a **microfp_s**
`--` Calls **subMicroFP()** to subtract 1.0 from a **microfp_s** value and return a **microfp_s**

Negation Operator:

- Calls **negMicroFP()** to negate a **microfp_s** value and return a **microfp_s**

Example of How your Functions are Called

You will only be writing the functions that are being called here (**toMicroFP, addMicroFP, subMicroFP, mulMicroFP, toNumber, negMicroFP)**, but it is good to know what MUAN will be doing whenever you are entering expressions into the language.

Example Inputs:

3.14 + 1.5 This will first call **toMicroFP()** on 3.14 to convert it to a **microfp_s** type.
Then it will call **toMicroFP()** on 1.5 to convert it to a **microfp_s** type.
Then it will call **addMicroFP()** on those two values to add them.

-2.0 This will call **toMicroFP()** on -2.0 to convert it to a **microfp_s** type.

print((1 + 2.5) - 3) This will first call **toMicroFP()** on 1 to convert it to a **microfp_s** type.
Then it will call **toMicroFP()** on 2.5 to convert it to a **microfp_s** type.
Then it will call **toMicroFP()** on 3 to convert it to a **microfp_s** type.
Then it will call **addMicroFP()** on the first two values to add them.
Then it will call **subMicroFP()** on that result and the value representing 3.
Then it will call **toNumber()** on that result and print out the final value.

2. MUAN Programming Language Scripts

You can run a script, like with python, by using the filename as a command line argument.

```
./muan scripts/sample.muan
```

Here is an example MUAN script in (scripts/sample.muan) with one statement per line:

```
# Sample Script
first = -0.45
print(first)
second = 4.5
print(second)
third = -0.25
fourth = third++
print(third)
print(fourth)
fourth += 1.5
print(fourth)
sum = first + second
print(first)
difference = first - second
print(difference)
product = first * second
print(product)
complex = 3 + second - 1.25 * -2
print(complex)
display(complex)
quit
```

Here is the output for the provided sample script.

```
kandrea@zeus-2:handout$ ./muan scripts/sample.muan
Welcome to the Micro-Ubiquitous Accounting Notary (MUAN) programmable calculator.
¯\_(ツ)_/¯ $ # Sample Script
¯\_(ツ)_/¯ $ first = -0.45
¯\_(ツ)_/¯ $ print(first)
first = -0.4453125
¯\_(ツ)_/¯ $ second = 4.5
¯\_(ツ)_/¯ $ print(second)
second = 4.5
¯\_(ツ)_/¯ $ third = -0.25
¯\_(ツ)_/¯ $ fourth = third++
¯\_(ツ)_/¯ $ print(third)
third = 0.75
¯\_(ツ)_/¯ $ print(fourth)
fourth = -0.25
¯\_(ツ)_/¯ $ fourth += 1.5
¯\_(ツ)_/¯ $ print(fourth)
fourth = 1.25
¯\_(ツ)_/¯ $ sum = first + second
¯\_(ツ)_/¯ $ print(first)
first = -0.4453125
¯\_(ツ)_/¯ $ difference = first - second
¯\_(ツ)_/¯ $ print(difference)
```

```
difference = -4.875
¯\_(ツ)_/¯ $ product = first * second
¯\_(ツ)_/¯ $ print(product)
product = -2.0
¯\_(ツ)_/¯ $ complex = 3 + second - 1.25 * -2
¯\_(ツ)_/¯ $ print(complex)
complex = 10.0
¯\_(ツ)_/¯ $ display(complex)
MicroFP Value in Binary: 0 110 01000 (0x0c8)
¯\_(ツ)_/¯ $ quit
SIC PARVIS MAGNA

Have a nice day!
```

3. MUAN Programming Language Interpreter

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

```
kandrea@zeus-1:handout$ ./muan
Welcome to the Micro-Ubiquitous Accounting Notary (MUAN) programmable calculator.
¯\_(ツ)_/¯ $ foo = 0.25
¯\_(ツ)_/¯ $ bar = -1.5
¯\_(ツ)_/¯ $ print(foo * 2 + bar)
Value = -1.0
¯\_(ツ)_/¯ $ print(-inf)
Value = -Infinity
¯\_(ツ)_/¯ $ display(-inf)
MicroFP Value in Binary: 1 111 00000 (0x1e0)
¯\_(ツ)_/¯ $ print(0 * inf)
Value = NaN
¯\_(ツ)_/¯ $ quit
SIC PARVIS MAGNA

Have a nice day!
```

4. MUAN Operator Precedence

Precedence is designed the same as you would expect in Python, C, or Java:

Highest Precedence

Parentheses and Increments	**( )**	**++x**	**x++**	**--x**	**x--**
Negation	**-x**				
Multiplication	**x * y**				
Addition	**x + y**				
Subtraction	**x - y**				
Assignments	**x = 3**	**x+=3**	**x-=3**	**x*=3**	

Lowest Precedence

5. Notes on MUAN

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

```
¯\_(ツ)_/¯ $ foo = 3.5
¯\_(ツ)_/¯ $ print(foo + 5.6 + (bar = 2.5))
Value = 11.5
¯\_(ツ)_/¯ $ print(bar)
bar = 2.5
¯\_(ツ)_/¯ $ 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 MicroFP value.

Since every expression results in a MicroFP value, by typing in 3.5 MUAN will convert it using **toMicroFP** while processing that statement.

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

• **inf**	Infinity
• **nan**	NaN

Make sure your **toMicroFP** can handle **inputs** of infinity (number->is_infinity == 1) and NaN (number->is_nan == 1) on inputs. Note that these two members only tell you if the user literally typed in "**inf**" or "**nan**" on input in MUAN.

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

Note: You can have is_infinity == 0 AND toMicroFP return infinity if the value was too big.

Example of using inf and nan in MUAN:

```
Welcome to the Zeus User Operations Notary (MUAN) programming language.
¯\_(ツ)_/¯ $ a = inf
¯\_(ツ)_/¯ $ print(a)
a = Infinity
¯\_(ツ)_/¯ $ a = nan
¯\_(ツ)_/¯ $ print(a)
a = NaN
¯\_(ツ)_/¯ $ a = -inf
¯\_(ツ)_/¯ $ print(a)
a = -Infinity
```

6. Changelog

v1.0: Sep 19: Release (Build: **2910c72a** in develop)