mo

od

yP

A dynamic, relatively insane programming language

moodyP: Documentation

Prepared for: Dr. Bilal Shibaro, COSC 3336.01 Programming Languages

Prepared by: Eduardo Perez && Liam Khuen, Students

November 26, 2016

Proposal number: 01

WHY MOODYP?

Welcome! At the moody Foundation, we believe that sometimes coding can be a little *too* simple (easy) sometimes. What happened to the days of unreliable punch cards and those little math machines that broke all the f****** time?

1.1.1 WAIT SO WHAT IS MOODYP…?

moodyP is a very-angsty programming language (some people in industry call it a “VAP-language”). A nice way to think of *moodyP* is to liken it to a nice broth-y soup:

“let me not be accounted for the taste of my soupiness,

but for the appearance of my broth

and whether or not it will give me

food poisoning”

-unknown

Ah, yes, the old *soup* metaphor! *moodyP* essentially takes all the power and graft of other popular scripting languages and very tightly packages their strength (strengths?) into a delicious bouillon cube of gooey goodness that, much like a person, may or may not obey your every command. See? Cool!

More on this later.

1.1.2 THIS IS ANNOYING AND I WONT LIKE LEAR–

WOAH WOAH there, cowboy! Why, you don’t even know any of the syntax or other specially scrumptious aspects of the moody Foundations’ pride and joy of a language? Just settle on down and maybe have a smoke. Or not. We at the moody Foundation, like, don’t really care what you do with your body, man. Just chill out and ride the wave 😈.

1.1.3 THE MEAT OF THE SOUP

```
moody()
```

is a “class”. In the traditional world of computing, a class is an abstract concept (that miners have been digging at for nearly 170 years whilst subsequently trying to find a more suitable name for this darn idea) that defines a structure which represents a container of *things*. What kind of things? Well:

things like *var(s)* which store data

things like *function(s)/procedure(s)* which do things with that data

things like *other stuff that we can talk about later since the two previous things are mostly used anyway*

Nah, just kiddin’. Classes actually can do quite a lot. For example…

1.1.4 FOR EXAMPLE

Let’s say you’ve got a nice, small dog. Now, this dog never growls, it never bites, but sometimes it likes to walk and eat.

```
dog = class
```

Since this dog sometimes likes to eat a bit, it will have a tiny little program in its avocado-sized brain that tells it to do so:

```
Procedure Eat()
```

Nice! Now you’ve got a normal dog. Yep, just a normal dog with a brain that can only tell it to do one thing.

Cool, so what we just created was something called a *subroutine*. Spell it with me now: S-U-…eh, sounds like you’ve got it.

1.1.5 WAIT WHAT’S THAT ONE THING BEFORE THE “EAT” PART

Looks like we’ve got an observer over here! Yes, right before the **name** of a Function/Subroutine/Method (yes, these are all names used interchangeably and yes they all mean the same thing) you must describe what the hell it is first.

1.1.6 COOL, I KNOW EVERYTHING ABOUT MOODYP!

Lol, no. Not so fast. You have to know about a bunch of other stuff too. Stuff like *Functions* which are similar to *Procedures* except that they are supposed to bring something back after doing stuff.

```
Function Eat(params...) : Return_Motion
```

What's that you say? *params? Return_Motion?* Alas, this is where the fun begins! You see, *moodyP* allows you, the soup-maker to create *Procedures* or *Functions* that take in certain inputs. Let's say your dog eats only the finest rubbery boots (a prized item in the economy of *moodyP-landia*).

```
var food : String;
food = "RubberBoot";
Function Eat(food) : Return Motion;
```

If all goes well within the little *Function* we wrote, the action of your dog eating *food* (a *"RubberBoot"* type of thing) shall yield you something called *Return_Motion* whatever that does.

1.1.7 SHOOT NOW WE GOTTA WORRY ABOUT FLOW

Why yes, We are dealing with logic, after all.

```
if [some kind of check on a var]
        writeln("Yum. . .I mean woof");
        [do stuff];
elsif [some like other condition]
        writeln("Eww, lmao Im a dog");
else
        Bye();
```

else sets an additional check for the data coming into the *Function*. Keep in mind, that you MUST use **elsif** before using an **else** check if you want to make more than two conditional tests for the data coming into the function. In this case, if the *Eat Function* does not receive the necessary input after a third test, the ***Dog*** will perform the actions described in ***Bye***.

1.1.8 MATH, TOO

moodyP loves math. As a matter of fact, most programming languages do. Everyone knows (we hope!) how to add, subtract, multiply, and divide. Here's how you do it in *our* language:

```
{ addition }
4 + 3
{ subtraction }
4 - 3
{ multiplication }
4 * 3
{ division }
4 / 3
```

1.1.9 PEMDAS PEMDAS PEMDAS PEMDAS

No, *moodyP* may not have any broads in Atlanta, but it sure does have support for parenthetical statements!

```
1 + 5/9 * (2+(10 - 1)) -2
{ See, wasn't that lovely? }
```

1.2.1 ERRORS AND ERRORS GALORE

Let's get this out of the way right now: you will get a ton of errors while you program using *moodyP*. You will try and try and pray and pray but the truth is your code will not or ever work. Well, sometimes it will. Mostly it wont.

moodyP handles errors with simplicity and finesse. That is, it has a very *rudimentary* error reporting system built-in. "Well, hot dog!", you scream in excitement. Let's say you try to be a bad little programmer and try to utilize an *INVALID* character in your code. Well, *moodyP*, being moody and all, will scream and kick and yell internally but will tell you this:

```
>> "Invalid character"
```

What line is that error on? Who knows! You see, half the fun of programming is learning how to suffer.

1.2.1 THE CRUX OF THE LANGUAGE

Okay, so remember when we spoke earlier about wanting to come back to a time in which computers were unpredictable? Well, that's the whole point of *moodyP*. In fact, the *interpreter* (basically, the thing that runs the show in the background; you don't really have to know how this stuff works) only runs when it **feels** like running.

1.2.2 MOOD LEVELS

Much as in the religion of *Scientology*, the *moodyP* programming language has certain layers of status. At runtime (a fancy word for when "stuff happens"), the interpreter decides upon a specific mood from a set. Just like a petulant child, the program will either happily scurry off with your code or curse your name under its breath for all eternity.

```
GOODVIBES()
{ Happy, content, calm, and excited }
BADVIBES()
{ Mad, upset, angst-y, and sad }

    { NOTE: The interpreter will print
   specific statements to the console at
    runtime depending on the internally
              selected mood }
```

1.2.3 SO, WHAT'S NEXT?

Wew, wasn't that just a bundle of fun? We here at the *MOODY FOUNDATION©* would like to ensure that the BUCK DOES NOT STOP HERE. There will be an attempt made to ensure that the greatest in operations can be made with this new type of VAP-langauge in the future, so stay tuned and gather your closest friends. You're gonna need 'em.

1.2.4 ACKNOWLEDGMENTS AND OTHER COOL STUFF

The totally-not-made-up *MOODY FOUNDATION©* would like to thank Professor Bilal Shibaro for being so cool so as to allow us to create such a monster. No, he did not force us to make this.

listOfKeywords:

```
var - declare variables
if/then/else/elsif - conditionals
true/false - boolean
string - collection of chars in a variable
array - multiple elements of the same type
+ - * / - math
```