## *Learn you a Yeet for great good*

So you want to learn Yeet? I mean, that must be why you're reading this. Stop. Think to yourself *where did I go wrong? Were my parents right? When will my dad ever say he's proud of… oops that slipped into my problems there—wait a second*. So you want to learn Yeet. Well, you've come to the right place, I don't think there's anywhere else that'll do it, especially just covering the subset of the language that has been implemented in time for the Friday submission, rather than the Wednesday presentation. So here we go.

**Values**
So what are we dealing with in Yeet? Well, there are two basic types in Yeet: Text and Number. *what about functions?* Three basic types in Yeet: Text, Number and Function. They all have quite obvious meanings for what they are, their names were chosen to be as simple to understand for youngsters *(that makes me sound like a grandad, doesn't it)* as possible. There is also another type, Device, which we've created for this project especially which represents an input or output device on the Raspberry Pi. All values are handled the same, however, there is no way of referencing a function without calling it (except as passing it as an argument to another function) due to the simple calling syntax which we'll cover later.

```
3       -- pi
"Hello" -- some text
6.28    -- tau
```

Numbers and text can both be created with literals, and we'll come on to creating functions a little later. Devices can also be referenced directly through something known as magic literals, which must start with a capital letter.

```
ButtonA -- both of these reference devices attached to the Pi
LightC
```

**Variables**
As with most languages, variables, the good old *x* and *y*s, work as expected. To simplify the current implementation, variables can only contain letters. Also due to magic literals (see above), they cannot begin with a capital letter. *For any of you regex fans, a variable identifier must match /[a-z][A-Za-z]*/.* This is hoped to be expanded by the final version to allow numbers after the starting character. Yeet is a lexically-scoped, dynamically-typed language. That just means names only exist in the block of code you define them in, you can put any kind of data anywhere. You must declare all names before you use them, but once declared you may assign a variable to whatever you want. To declare a name, just `let` *`it`* be *(use the let keyword, then write the name of it)*. There is no *null* or *undefined* type in Yeet right now, the thinking being to make it is a simple as possible for kids to use as possible, any function that'd you think should

return *null* returns *0*, although this may change. Thus, when you declare a variable, you *must* assign it to something.

```
let x = 0
let name = "Jonover Simpletonson"
x = 3
-- let name = "Woot"  would now be an error, as `name` has already been defined
-- let y  would also be an error, as you must assign `y`
```

**Functions**

Now we get to making stuff that's actually worthwhile, and for that we'll need functions. Yeet really puts the *fun* in *fun*ction, by making them simple to define, and easy to call. One of the main objectives of yeet was to minimise the amount of syntax, especially symbols, so it would be much easier to pick up, and adults would get less pesky questions about what `(\f -> ($) (f .).f)` means *I still love you Haskell okay, it's just, think of the children. Talking about children, have my babies please Haskell.* To call a function in Haskell, sorry I mean Yeet, just put the name of the function followed by the arguments.

```
function first second third -- given all is defined
say "Hello" -- Currently `say` is the only built-in function, it prints to stdout
noArgumentFunction -- This will call the function,
                   --  hence functions can't be directly referenced
```

To define a function, we need to declare the name and tell Yeet what to `do` when it comes across it. We give it a simple list of instructions, and Yeet'll do them in order. Note that functions are closures, they keep a hold of all the names on the stack when defined (not the values, those use the latest values). In a function declaration, the first identifier is the function name, and the following are an argument list, much like ~~Hask~~ function calls.

```
let x = 0
let messWithX newValue do
    say x
    x = newValue

messWithX 1 -- says 0
say x       -- says 1

let dontMessWithX newValue do
    let x = newValue
    say x

dontMessWithX 2 -- says 2
say x           -- still says 1
```

Well, that's handy, especially if given functions like `shine` which would let you turn a light on. But wait, there's more! *I'd hope so, so far it hasn't impressed me*. The last line executed of any function is the value returned by the function *(much like Ruby, Crystal, ...)*. We hope to introduce a `return` statement eventually that you can use to explicitly return a value. You can also define multiple implementations of a function (overloading), pattern matching *sort-of* on the arguments provided by writing literals instead of identifiers. Any kind of literal, text, number, or magic, can be matched against. You can even redefine the function given the same arguments if you want to (overriding).

```
let left first second do
    say first
    first

left first 0 do -- You've already declared this, no need for `let`
    say "You gave me 0 on the right because you think it's pointless?"
    say "Well I'll show you"
    0

let x = left 3 1
say x -- says 3.0
x = left 3 0
say x -- says 0
```

Note, like other values, you can define a function in a function, and even pass a function to another function.

```
let apply function value do
   say function
   say value
   function value

apply value do
   let apply do
       say "apply with only one argument"
   apply
   value

let result = apply apply 5
say result

{- Outputs:
    Function<defs:2>
    5.000000
    apply with only one argument
    5.000000 -}
```

Once the event handling is implemented, all events will have a dummy function implemented in the language which does nothing, which you can overload for each device. This will be called when the event is triggered.