JythonC

A simple yet typed programming language designed to be reliably easy to program in.

Logan Farmer

Theory of Programming Languages
Marist College
Dr. Labouseur
04-27-22

1 An Introduction

If you weren't able to tell from the name, JythonC was inspired by the languages Python, JavaScript, and C/C++. While these are all decent programming langugages on their own, taking the best parts from each would help create a language that's easy to learn and write like python, but with all the typing and functional help from JavaScript and C/C++. This language excels at dealing with large decimal numbers and scientific math operations run on datasets.

There are pros and cons to having your language being on either end of the "typed" spectrum (like how far Python is from Java), and it really depends on both the coder and the project getting done on how much of an impact those pros and cons have. In JythonC, here's what's staying and what's going from the languages it's based on:

STRONG typing In a language like Java or C++, if you were trying to add an Int to a Float, it would convert the Int to a float, and then do the addition. This causes more work for the compiler to allocate enough space to store the float in memory. While this might save space in the long run, it takes away from readability and writability. So in JythonC, each data type that's being operated on must match according to the typing rules. However, you are able to cast one data type to the next, with an argurment to choose what type the result to be.

Numerical Type Every type of number shares the same type in JythonC; integers, floats, doubles, etc... are all reffered to as "Numbers." Declaring a number is as easy as setting the type to number like:

```
myValue:Number() = 3
```

The default type is an integer, and you are able to create more specific numbers like this:

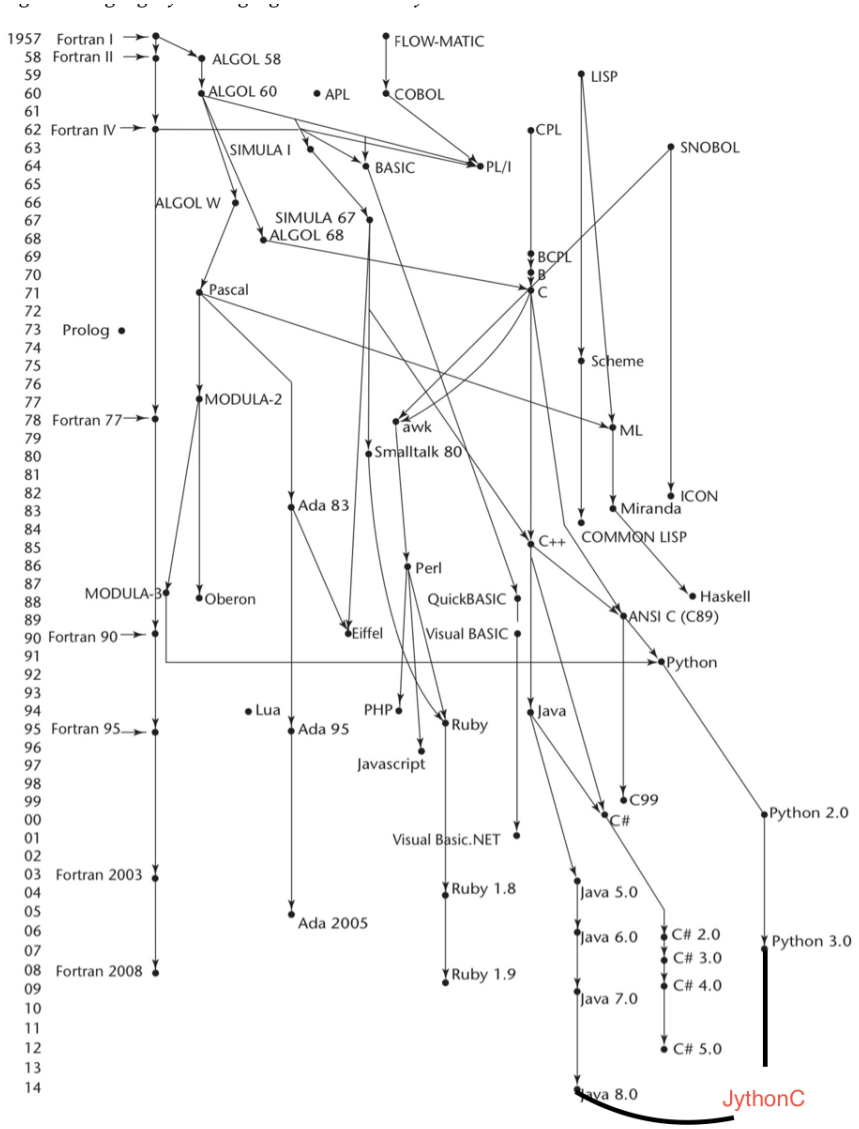
```
//double for two places after the decimal
myCurrency:Number(Double) = 142.52
//decimal for an extended decimal values, second parameter is how many digits max
myScientficValue(Decimal, 10) = 0.1589105145
```

Traditional classes and objects? None here. Each file you create becomes it's own "Module" with a name that you can use to reference it that's the same as the filename. So if you created a file called Math.jyc containing methods for all the basic arithmetic operations (+, -, *, /), you'd be able to use them in another file by importing them to your current module. You'll be able to represent real life entities as an object that's a hybrid between a JSON object and a Java Class. More on that below.

Lambda functions While this isn't new to either language, the use of lambda functions wouldn't be the first thing that comes to mind when you think of Java or C (Python is a different story). The syntax for a lambda expression in JythonC looks like this: (args) ==> { code block }. The semantic value of this expression will be equal to the result of the expressions contained in the code block. After seeing how to declare functions coming up, you'll see how similar this syntax is to just using a normal function.

Program structure As you'll see shortly, these aren't the only things that are different between Java, Python, C/C++, and JythonC. Python's indentation formatting was removed so there's a good mix of readable keywords and control structures. Similar to Java and C, the use of the braces ({}) indicated a new block of code and a new level of scope.

1.1 Genealogy



You can see from this genealogy chart that JythonC can easily be called a "Modern" programming language. From the very beginning of this chart, you can see the evolution of programming languages slowly making it's way to what is today considered "The Big Three" which is Java, Python, and C (as of 2014).

1.2 Hello World

```
hello: (name:String) ==> {
    print("Hello " + name + "!")
    println("With love from JythonC")
}
hello("Logan")
```

1.3 Program Structure

Writing a program in JythonC will feel more like coding in JavaScript or Python than it is in Java. There's no main function or class, all you need to do is type some code and send a file to the compiler. In general, it's best to organize your files (or modules) by what their methods accomplish. Here's an example of two ways to organize using simple math:

program.jyc

```
power: (val:Number(Integer), Exp:Number(Integer)) ==> {
    num:Number() = val;
    for(0, exp) {
        num = num * val
    }
    return num
}

main: () ==> {
    print("2 to the 4th power is equal to: " + power(4, 2))
}

main()
```

And here's an example of a similar program, this time with the use of modules:

math.jyc

```
factorial: (val:Num(Integer)) ==> {
    val > 0 ? return val * factorial(val-1) : return 1
}
```

app.jyc

```
use factorial from "./math"

println("Factorial of " + 3 + " = " + math.factorial(3))
```

1.4 Types and Variables

There are only a few data types, and only two of them are primitive types: increment variables (with a max of 4 bytes), and characters. Every other data type (string, numerical) are represented by reference variables, which point to more discrete areas in memory where precise information can be stored. You can assign any number of variables (as long as you haven't run out of possible variable name combinations that have less than 32 characters) to reference the same object in memory.

1.5 Statements Different from JavaScript and Python

1.6 Object-Oriented

This language differs the most from Java and Python with the removal of classes. There's a built data structure similar to an object in JS or a dictionary in Python. This consists of key value pairs, which is formatted correctly can have the same effect as an object, just minus the methods. Here's an example of how I would represent an animal in a Java-like language vs JythonC:

```
public class Animal {
    private String name;
    private int age;
    private String type;
    private String breed

    public Animal () {
        this.name = "";
        this.age = 0;
        this.type = "";
        this.breed = "";
    }

    public Animal(String type, String name, int age, String breed) {
        this.type = type;
        this.name = name;
        this.age = age;
        this.breed = breed;
    }

    ..... (20-30 lines of getters and setters) .....


}
```

This is a lot of lines of codes to be writing for a programmer who if they know what they're doing, can manipulate this same data without having to use such an abstract representation. Here's how it would look in JythonC:

```
myAnimal = {
    name: "Andy",
    age: 2,
    type: "Cat",
    breed: "Persian"
}


print(animal.age) //2
animal.age++
print(animal.age) //3


\\built in toString() function to output contents of objects
print(yourAnimal.toString()) //undefined (doesn't exist yet)
yourAnimal = myAminal
print(yourAnimal.toString) // name: "Andy", age: 2, type: "Cat", breed: "Persian"


//use the fresh function to make a copy of an object with only the keys, not the values
yourAnimal2 = myAnimal.fresh()
print(yourAnimal2.toString()) // name: null, age: null, type: null, breed: null
```

As you can see, this method is less reliable, but is much more writable than a java class. There area also ways to use class methods, and it's slightly less intuitive but once you take a look at it it makes sense.

```
myAnimal = {
    name: "Andy",
    age: 2,
    type: "Cat",
    breed: "Persian"
    birthday: () ==> {this.age++},
    ageAfter: (years:Num(Integer)) ==> {return this.age + years}
}

print(myAnimal.age) //2
myAnimal.birthday()
print(myAnimal.age) //4
print(myAnimal.ageAfter(10)) //14
```

This is a simple example but the same logic can be applied to more complicated examples, which you will see below. In this specific example, the function is able to access the class variables using the this keyword. You're only able to access these without passing in the class variables because the function was declared inside of the object as a lambda function.

2 Lexical Analysis

2.1 Running Programs

A program is run from any text file that is sent to the compiler with a .jyc file extension. There are no pre-defined structures like packages in java to help organize your code. As long as there are correct references to other files (or modules) on the same machine the main file is being run, there will be no issues with someone severely disorganizing their code. While this is makes JythonC less reliable, it's arguably much more readable and writable than using a package system such as Java. Here's what an example file structure for an application might look like:

```
-src
--main.jyc
-+util
-----AsciiFormatter.jyc
-----dateManager.jyc
-+data
-----clientConnect.jyc
-----http.jyc
-+service
-----service1.jyc
-----service2.jyc
-----service3.jyc
```

Listing 1: File Organization

In order to use these other services and utility function in main, you'd need to import them similar to JavaScript or Python.

main.jyc

```
import * from "./service/service1"
import method3 from "./service/service2"

//if you imported "*", you can call a function from one module like it's in the
//same module you're working on
method3()

//if you didn't use the "*", you'll have to specify which module you'd like to
//use a function from
service2.method3()
```

Listing 2: Importing Modules

2.2 Comments

Comments work similarly in JythonC as they do in JavaScript. You can use "//" to start a single line comment; Everything you type after these forward slashes will be a comment until you start a new line. You can also create multi-line comments by starting the comment with "/*" and ending it with "*/". Everything contained inside these two tokens will be considered part of the same comment. These are called delimited comments. Feel free to write an entire novel contained within these bounds.

2.3 Production Differences

A good example of where the production rules might differ when the language is written in BNF is when it comes to importing modules and packages. Here's a snippet of how Java does it:

```
<import declarations> ::= <import declaration>
                        | <import declarations> <import declaration>

<import declaration> ::= <single type import declaration>
                        | <type import on demand declaration>

<single type import declaration> ::= import <type name> ;

<type import on demand declaration> ::= import <package name> . * ;
```

Here's how that would look in JythonC BNF:

```
<import declaration> :== import <methodList> from <module>
<methodList> :== <method> <methodList> | * | epsilon
<method>     :== <method name>
<module>      :== *filepath pattern*
```

2.4 Keywords

There are a few *key* differences between the keywords in Java/Python vs JythonC. The first being the removal of the important class keywords and visibility keywords that don't apply to JythonC, such as:

- class
- super
- extends
- implements
- interface
- private, public, protected, void
- package

There are also several python keywords that I'm not going to include in this document for the sake of space. Some of these include "and", "none", "or", "def", or "is". I can see why it's helpful for readability and writability.

3 Type System

JythonC uses a strong static type system. Errors will be caught in compilation, and once a variable has been declared as a certain type it will remain that type. There are two categories of types: value and reference types.

3.1 Value Types

- Number(Integer): a number without decimals.
- Char: a single character.
- Boolean: A type that is either true or false.

3.2 Reference Types

- Number(Decimal): A number that has a set number of places before and after the decimal point.
- Array: a collection of data with a set type and length
- List: a collection of data with a set type and mutable length
- Dictionary: a set of key value pairs that do not have any type restrictions. Any data can be a key, and any data can be a value, including lambda functions, numbers, and strings. This can be used to construct class like representations.
- String: An array of characters

4 Example Programs

4.1 Caesar Cipher Encrypt & Decrypt

Both decrypt and encrypt can be done using this method. When you call the function to encrypt, you pass the shift value in as the second parameter. To decrypt it, all you have to do is pass in (26 - shift value) as the second parameter.

```
encrypt: (message:String, shift:Number(Integer) ==> {
   return message.map((c) ==> {
    return c.isLetter() ? ((ord(c) - ord('A') + shift) % 26 + ord('A')).toChar() : c
   })
}
```

4.2 Bucket Sort

Bucket sort is used to further the effectiveness of another sort (quickSort, mergeSort, etc...) when used over a uniformly distributed set of data by bunching together groups of similarly valued numbers in linear time. This method has a higher space complexity than just using another sorting algorithm on the full dataset, but becomes faster than normal sorting the closer the data gets to perfectly uniform.

```
BucketSort: (nums:Array) ==> {

   k:Number() = sqrt(nums.length)
   max:Number() = max(nums)
   index:Number() = 0
   buckets:Array(k, List(Number(Integer))) //create an array of k buckets

   for(i in nums) {
      index = floor((nums[i] * k) / max)
      buckets[index].add(nums[i])
   }

   for(i in range(0, k)) {
      sort(buckets[i]) //call built in mergesort or call your own to sort each list
   }

   return buckets.concatenate()
}
```

4.3 Factorial

This was an example used above but here it is again. This is a good use of the ternary operator, which evaluates the first boolean expression and then executes the second statement if it's true. Else, it will execute the third statement that comes after the colon.

```
function factorial = (val:Num(Integer)) ==> {
    val > 0 ? return val * factorial(val-1) : return 1
}
```

4.4 Fun with Lambdas

Here's an extreme example of how you can use lambdas inside of lambdas inside of lambdas which are inside of AND referencing objects.

```
import pricePerGallon from "gas_server"

myCar = {
    make: "Toyota",
    model: "Prius",
    year: "2010",
    tankSize: 9
    costToFill: (litersLeft:Number(Double) ==> {
        //both lambda function is entirely unnecessary but IT CAN BE DONE
        return (() ==> {return this.tankSize * 3.785} - litersLeft) * () ==> {
            return pricePerGallon() * 3.785
        }
    }
}
```

4.5 Stack

```
stack = {
   max: 20,
   values: Array(max, String),
   top: 0,
   isEmpty: () ==> { return top < 0 },
   isFull: () ==> { return top == max },

   push: (s:String) ==> {
      if(this.isFull()) {
         return false //stack overflow
      }
      else {
         values[++top] = s
         return true
      }
   },

   pop: () ==> {
      if(this.isEmpty()) {
         return "ERROR - NO VALUE"
      }
      else {
         return values[top--]
      }
   },

   peek: () ==> {
      if(this.isEmpty()) {
         return "ERROR - NO VALUE"
      }
      else {
         return values[top]
      }
   }

}
```