Attempt

Dylan Johnson

May 3, 2016

Introduction

This is another attempt to write a language; this one will compile for the JVM. I decided to compile to the JVM so that a lot of difficult details, such as garbage collection and heap allocation can be abstracted away. I may change the target the environment to LLVM and write my own garbage collector in the future. However, that is just speculation at this point. Lets get things done one at a time. It is going to be named Attempt because that is likely all this will turn out to be. In this document will reside the specification for the language.

Feature Plan

Current Plan

- enforce proper whitespace indentation and formatting
- all the standard control flow tools; loops, if else blocks, ect
- classes with internal member variables and methods; polymorphism via interfaces and data encapsulation
- add the basic built in types, strings, arrays, integers, bools, floats, ect
- strong type system
- all the proper operators for builtin types; +, -, =+, [], ect
- scopes
- compile to executable class files
- a proper namespace and module system

And Beyond

- generics
- port to LLVM
- integration with Java stdlib
- write a garbage collector
- stdlib
- switch to using impractical unicode symbols such as λ or $\rightarrow$

Specification

High Level Formatting

Whitespace

The only valid whitespace is the one true whitespace: spaces. Spaces and only spaces will be allowed. This is because spaces are the purest of all whitespaces and they stay firm to their indentation from one editor to the next. All indentation must be in multiples of a given power of two; in other words, you cannot mix indentation that indents by one space or two spaces. If you choose and indentation of one space, you must stick with it or you will suffer. Your program wont compile.

Statements

Each line of code makes up a statement that can contain one or more expressions. Each expression is delimited by a semicolon. The end of each line cannot end with a semicolon.

```
# the following expressions are equivalent
let x: i32 = 3; x += 1
# or
let x: i32 = 3
x += 1
```

Code Blocks

The : symbol will be used to state that a new code block will be starting on the next line. The : symbol must be used in conjuction with specific language constructs. For example, defining the body of a method, if statement, or for loop. They can be nested and can be use in a solitary fashion as well. When used in a variable declaration, it is used to specify the type. Each block defines its own lexical scope.

```
if <expression>:
  <expression>
else:
  <expression>

# the for loop contains 3 expressions
for let x: i32 = 0; x < 20; x += 1:
  <expression>

func foo() -> i32:
  <expression>
  return 3

<expression>
: # defines a new block.
  <expressing>
```

Operators

assignment =

add assignment +=

subtract assignment -=

multiply assignment *=

divide assignment /=

mod assignment %=

binary negate assignment ∼=

binary right shift assignment >>=

binary left shift assignment <<=

binary and assignment &=

binary or assignment | =

binary xor assignment ˆ=

add +

subtract -

multiply *

divide /

mod %

array index [ ]

logical and &&

logical or ||

not !

equivalence ==

not equivalence !=

less than <

less than or equal <=

more than >

more than or equal >=

binary negate ~

binary right >>

binary left <<

binary and &

binary or |

binary xor ˆ

scope resolution ::

comment #

Primitive Types

8 bit signed integer i8

16 bit signed integer i16

32 bit signed integer i32

64 bit signed integer i64

8 bit unsigned integer u8

16 bit unsigned integer u16

32 bit unsigned integer u32

64 bit unsigned integer u64

32 bit IEEE 754 floating point f32

54 bit IEEE 754 floating point f64

a single byte byte

a 18 bit unicode character char

a true/false boolean bool

a string of characters. internally handeled as an array of chars string

Functions

func

The *func* keyword is used in function definitions.

```
# returns nothing
func foo():
  <expressions>

# returns nothing takes a i32 parameter
func bar(i: i32):
  <expressions>

# takes a f32 parameter and returns a f32
func baz(i: 32) -> f32:
  <expressions>
  let r: f32 = 0
  return r
```

Functions do not need to be defined in a *class* definition.

entry point function

In each executable program, there will be one and only one entry point method. This method cannot be defined in a *class* definition and must reside in the root *namespace*. It will have the following signature. The *args* parameter are the command line arguments.

```
func main(args: []string):
  <expressions>
```

Object System

interface

The *interface* keyword is used to define the beginning of a interface definition.

```
# beginning of a interface
interface IBar:
  func Baz(x: i32) -> u32
  func Bar() -> i32
```

All methods defined in a interface are public when implemented they cannot be private.

class

The *class* keyword is used to define the beginning of a class definition.

```
# beginning of a class that implements the IBar
# interface this class is public; this means that
# it is exported outside of it's module
# if the public keyword was not present in its
# declaration, it would only be accessible inside
# it's module and modules that are lower in the
# heirarchy
public class Foo(IBar, <interfaces to implement>):
  # private member data definitions
  x: i32
  y: u32
  # public member data definitions
  public z: f32

  # the empty constructor
  func Foo():
    x = 0
    y = 0
    z = 0

  # constructor with parameters
  func Foo(x: i32):
    this.x = x
    y = 0
    z = 0

  # part of IBar implementation
  # public function
  public func Bar() -> i32:
    <expression>
    # each method in a class has a reference to the
    # instantiated object
    return this.x

  public func Baz(x: i32) -> u32:
    <expressions>
    return this.y + x

  # private function
  func Bin() -> f32:
    <expressions>
    return this.y
```

Modules and Namespaces

namespace

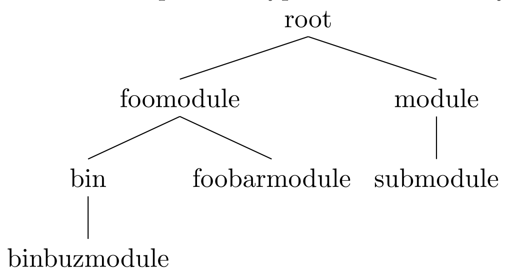
The *namespace* keyword is used to define the beginning of a namespace. Namespaces must be declared in the bottom of a file, ie, they cannot be declared in any class or function.

```
namespace foo:
  <class definition>
  <function definition>
  <variables>
```

All varibles that are declared in a namespace are constant. Global mutables are silly anyway. In addition to this, functions and variables can be exported with the *public* keyword. As in *classes*.

modules

Each *module* is composed of a subdirectiory, and subdirectories of the subdirectory, of the root directory. All source files in each directory make up the contents of the file and the normal export rules with public apply. The tree below is an example of a hypothetical directory heirarchy.



To access the exported contents of each module, you must descend into the module heirarchy and reference the class, function, interface, or constant variable expicitly using the *::* operator. There will be no “importing.”

```
# foo is in the root module
func foo():
  foomodule::bin::<function>

# bar is in the bin module
func bar():
  # must be in the bar module
  <function>
  binbuzmodule::<const variable>
```

A module that is lower in the heirarchy cannot reference a module that is higher than it. For example, no function in the *binbuzmodule* cannot reference anything in the *bin* module.

If you whine and say silly things like “but how will we import modules that are not in the project?” I say unto you, “grow a pair and use git submodules.” If that answer is not adequate for you then you are not worthy.

Features Implemented