Attempt

Dylan Johnson

May 2, 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
- integration with Java stdlib
- scopes
- compile to executable class files
- a proper namespace and module system

And Beyond

- generics
- port to LLVM
- 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 does not require 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 ^

scope resolution ::

comment #

Keywords

interface

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

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

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

class

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

```
# beginning of a class
class Foo(<interfaces to implement>):
  # private member data definitions
  x: i32
  y: u32
  z: Bar
  # public member data definitions
  public y: f32

  # public function
  public func Bar() -> i32:
    <expression>
    return this.x

  func Baz() -> u32:
    <expressions>
    return this.y
```

Primitive Types

Object System

Encapsulation

Interfaces

Namespaces and Modules

Features Implemented