The Pumpkin Programming Language

Joshua Boggs (jjb2175, Testing), Christopher Evans (cme2126, Architecture),
Gabriela Melchior (gdm2118, Manager), Clement Robbins (cjr2151, Language)

September 25, 2014

1 Introduction

The Pumpkin programming language is a lite-functional language, which allows for coding flexibility and concise syntax. This language focuses on easily modeling a complex system of function calls. The language would make it easy to model game-theory problems, as well as resolving theoretical and fantanstical rule-sets.

2 Syntax

2.1 Comments

Multi-line comments are written C-style.

```
/*
The slash and asterisk mark the beginning
and end of a multi-line comment.
 */
```

Single line comments are symbolized by two forward slash characters.

```
// Slashes mark the beginning of a single line comment.
```

2.2 Variables

Data Types

- Integer: declared Int
- Float: declared Float
- Double: declared Double
- String: declared String
- Boolean: declared Bool
- Unit: declared Unit

Immutable variables are declared with the 'val' keyword. They cannot be reassigned.

```
// Follows form val name: val = value

val aNumber: Int = 5
aNumber = 10
// Throws compile error: Cannot reassign 'val' declaration

val aNumber = aNumber + 5
// Throws compile error: Assignment on val cannot use a recursive assignment

val aNumber = 10 // No error thrown on redeclaration of a variable
```

Mutable data types are declared with the 'var' keyword.

```
var aNumber: Int = 10
aNumber = 5 // No error
```

Ideally, with an implementation of Hindley–Milner type inference, basic declarations of types should be unneccesary.

2.3 Conditionals

Logic expressions:

```
!True == False     // => True
not True == False // => True
False is False     // => True
True is False      // => False
True is not False // => True
True and True      // => True
True or False      // => True
not True           // => False
```

Logical Operators:

```
and   // Logical and
or    // Logical or
not   // Logical negation
```

2.4 Basic Data Types

2.5 N-tuples

Tuples are a basic immutable data structure that holds an ordered set of elements.

Unlike Lists or Maps, the elements with a tuple can be of any type and do not need to follow a consistent type declaration.

Tuples are most useful when a function may wish to return more than a single piece of information, or more commonly used as part of function nesting with 'pipe ins' and 'pipe outs'.

The following symbol scheme will be used to access consecutive elements of a tuple: `$0, $1, $2, ... $n-1`.

```
val t = (1, "hello", "world", 5)

t |> (x: Tuple => if ((x$0 + x$3) % 2 == 0) println("#{x$1} #{x$2}")) // Prints "hello
    world";

//Alternatively
t |> {
(x: Int, a: String, b: String, y: Int => if ((x + y) % 2 == 0) println("#{a} #{b}"))
}
// Prints "hello world";
```

2.5.1 Lists

Lists replace arrays as the basic iterable data structure. Lists are zero-indexed and immutable, so that all operations create new copies of the list. Lists accept any type but must have a consistent type across all elements.

Lists should also support basic List.head and List.tail features, as well as be able to be pattern matched in the follwing form: `head :: tail`.

The empty List also has a special type that can be used in pattern matching and other usefull circumstances called `Nil`

```
//Normal construction
val myList: List[Int] = List(1, 2, 3, 4)

// '::' is an operation that creates a new list by appending the element to the head
val newList = 10 :: myList // => List(10, 1, 2, 3, 4)

// You can use square-bracket "array" syntax as a shortcut for list creation

val arrayList = [1, 2, 3, 4] // => List[Int]: List(1,2,3,4)

myList match {
  case x :: xs =>
    println("Head: #{x}")
  case Nil =>
    println("Empty List")
}
```

2.5.2 Maps

Maps act as a basic Key:Value data structure that is staticly typed on both Key and Value. Maps are unordered and no guarantee can be made on the ordering of contents. Keys can be only primative types.

```
// Key-value pairs are inserted in the form of tuples
val myMap = Map(("x", "y"), ("a", "b"), ("t", "s"))

// There is also a shorthand for creating maps as followed:
val anotherMap = Map("x" -> "y", "a" -> "b", "t" -> "s")

val fetchedVal = myMap("x"); // => "y"

/*
You can get a List of keys and values from the map. However, nothing is guaranteed on
    the ordering of this returned List
*/

val keys = myMap.keys()
val values = myMap.values()
```

2.6 Functions

Functions can be defined with the def key word. The types must be specified.

```
def funcName(parameter: type): returnType = {}

// You can use 'def' to declare variables
// But unlinke 'var' and 'val' you do not know the value of 'x' until it is used
def x: Int = 5
```

For example:

```
// value is determined when x is evaluated in an expression
def x: Boolean = even(2)

val x = even(2) // value is determined at compile time
```

Anonymous functions are also allowed for flow control.

```
(variableName: dataType => code): returnType
```

2.7 Function Nesting

The symbols $| >$ and $< |$ can be used to chain nested function calls, the expression on the bar side is evaluated first. All arithmetic operators are applied before evaluating left to right, and parentheses are respected as in traditional order of operations. All expressions on the call side of the flow must be functions.

For example:

```
val a: Int = 3;
a |> (x: Int => x + 1): Int // Returns 4;
```

NOTE: The function calls funcName(x), $x| > funcName()$ and $funcName() < |x$ are semantically the same, but resolve differently with different precedence.
More examples of control flow:

```
b |> (x: Int => x + 1): Int |> even
```

The above expression gets executed as follows:

1. Evaluate expression b: Int => 3.
2. Left-to-right: pipe into expr2, an anonymous function which takes one argument (an Int), adds 1 to it and returns a new Int.
3. Pipe result into even(), which is a function that takes an Int and returns a Boolean.
4. No more pipes, left-to-right is done, return boolean value.

```
even <| (b |> (x: Int => x + 1): Int)
```

The above expression gets executed as follows:

1. Evaluate expression even: (x: Int => Boolean).
2. Pipe left, so look for return of Int from exrp2.
3. Expr2 evaluates to another pipe in parenthesis, so break it down and start left-to-right.
4. Evaluate expression b: Int => 3.
5. Pipe right into anonymous function that returns an Int.
6. Going back to original Expr2, we have the return of Int, we can now pipe it back into Expr1 which is (Int => Boolean) 'even' function.
7. After evaluating there are no more pipes or operations, return boolean.

```
even <| b |> (x: Int => x + 1)
```

The above returns type error, since expr1 is evaluated into 'even < | b' and expr2 is '(x: Int => x + 1)'. Since, without parenthesis, evaluation is left to right, this expression throws a type error.

2.8 Composing Functions

The << and >> operators can be used to call a fuction with the return type of another. For example:

```
(f << g) <| x or x |> (f << g) // is same as f(g(x))
(f >> g) <| x or x |> (f >> g) // is same as g(f(x))
```

NOTE: If at any time the number of arguments for composed functions don't match/type wrong, compiler will throw errors.

Another example:

```
def timesTwo(x: Int):Int = x * 2
def plusOne(x: Int):Int = x + 1

def plusOneTimesTwo(x: Int): Int = x |> (plusOne << timesTwo)
def timesTwoPlusOne(x: Int): Int = x |> (timesTwo << plusOne)

plusOneTimesTwo(9) // => 20
timesTwoPlusOne(9) // => 19
```

2.9 Pattern Matching

The language also should have the capacity to do pattern matching

```
val a: Int = 5
val b: String = a match {
    | x:Int if (x < 0) => "Negative"
    | x:Int if (x is 5) => "Five"
    | _ => "Another Number" // Exhaustive case match
}
```

2.10 Escape Characters & String interpolation

We will use the same escape characters as Java, i.e. \n for the newline character. Additionally, the hash character '#' must be escaped.

The language also allows for the following style of string interpolation (like Ruby):

```
// using #{Expression} will evaluate everything within the curly brace
val name: String = "Jack"
val x: String = "There are #{1 + 4} apples, #{name}" // => "There are 5 apples, Jack"
```

2.11 Algebraic Data Types

Immutable data-holding objects depend exclusively on their constructor arguments. This functional concept allows us to use a compact initialization syntax and decompose them using pattern matching.

```
/** Below is an example of a prefix notation adder that evalues a single addition
or multiplication at a time
**/

//Base types act cannot be instantiated

base type Expr

alg Lit(num: Int) extends Expr
alg Binop(operator: String, left: Expr, right: Expr) extends Expr

def eval(expr: Expr): Int = {
  expr match {
        | Lit(n) => n
        | Binop("+", l, r) => eval(l) + eval(r)
        | BinOp("*", l, r) => eval(l) * eval(r)
        | _ => 0 // If doesn't match
    }
}

val onePlusTwo = Binop("+", Lit(1), Lit(2))

eval(onePlusTwo) // => 3: Int
```

3 Example Programs

3.1 Calculating GCD

```
def gcd(a: Int, b: Int): Int = {
  b match {
          | 0 => a
          | _ => gcd(b, a % b)
    }
}
```

3.2 Pre-Order Tree Traversal

```
type Tree(lft: Tree, rght: Tree, value: int)

def makeTree(value: Int): Tree = {
    if (value < 15)
        Tree(makeTree(value + 1), makeTree(value + 1), value)
    else
        Tree(Nil, Nil, value)
}

def merge(l1: List[Int], l2: List[Int]) = {
  l1 match {
          | Nil => l2
          | head :: tail => head :: (tail, l2 |> merge)
          |  _ => List()
  }
}

def preOrder(t): List[Int]{
  t match {
        | Nil  => List()
        | _ => {
        t.value :: (preOrder(t.lft), preOrder(t.right) |> merge)
        }
}

t = makeTree(0)
preOrder(t)
```

3.3 Fantasy Game

```
Actor(level: Int, hp: Int, defense: Int, attack: Int)

var dragon = Actor(20, 200, 20, 30)
var person = Actor(5, 50, 1, 10)

def attack(p1: Actor, p2:actor): Unit = {
    var rand = random()
    rand match {
        case rand > .5 =>
            p2.hp -= (p1.attack - p2.defense*rand) * rand * p1.level / p2.level
        case _ => Unit
    }
}

def battle(p1: Actor, p2: Actor): Unit = {
    while not p1.hp == 0 and not p2.hp {
        (p1, p2) |> attack
        (p2, p1) |> attack
    }
}
```