The Pumpkin Language Reference Manual

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

October 28, 2014

Contents

1 Introduction

Pumpkin is patchwork functional programming language. The Pumpkin programming language is a light-functional scripting language, which allows for coding flexibility and concise syntax. Pumpkin supports many syntatic optimizations for function nesting and chaining, such as pipes and partially applied functions. This language focuses on easily modeling a complex system of function calls.
The Pumpkin programming language compiles down to Java, thus all data types will compile to JVM compatible data types.

2 Types and Expressions

2.1 Naming

Variable, abstract data types and function names must be a sequence of letters, digits and underscores. The first character must be a letter. By convention we will use CamelCase for names with multiple words.

2.2 Variables

A variable is a storage location paired with an associated name and type, which contains some value. Variables cannot be cast to other variable types.

To convert between integers and floats use native API function asInt(float x) and asFloat(int y). Functions are first-class and can be passed, assigned and returned as variables.

2.2.1 Declaration

Individual variables must be declared one-per-line and must be initialized with a value. A declaration specifies a mutability indetifier, name, type, and value.

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

```
// Follows form val name: type = 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. They can be reassigned to different values.

```
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 Native Types

- **Int**: a signed two's complement integer, as in Java, which has a minimum value of -2^{31} and a maximum value of 2^{31}-1.
- **Float**: a floating point value that can represent either very small fractions or numbers of great magnitude.
- **String**: an immutable type that contains a sequence of characters.
- **Bool**: a boolean type whose only two possible values are True and False
- **Unit**: represents an output of nothing, similar to void in C and Java.

2.3.1 String

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"
```

Double quotes inside strings are not supported. Use single quotes instead.

2.3.2 Boolean

Boolean variables are either True or False. They can be manipulated with logical expressions and operators.

Relational Expressions:
Boolean variable can be compared and combined with logic symbols to form expressions. Equality tests can be written with the keyword **is** or with the symbol ==. Other comparisons use the standard symbols used in Java:

```
1 < 2     //Less than            => True
3 > 4     //Greater than         => False
1 <= 3    //Less or equal to     => True
2 >= 2    //Greater or equal to  => True
3 is 4    //Equality             => False
5 == 5    //Equality             => True
```

Logical Operators:
Pumpkin supports the three basic logic operators. Logical and can be written with the keyword **and** or with the symbol &&. Logical or can be written with the keyword **or** or with the symbol ||. Negation can be written as the keyword not or with the symbol !.

Examples:

```
False is False     // => True
True is False      // => False
True is not False  // => True

not True           // => False
!False             // => True
True and True      // => True
False && True      // => False
True or False      // => True
False || False     // => False

!True == False     // => True
not True == False  // => True
```

2.4 Derived Types

2.4.1 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";
```

2.4.2 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.

Pumpkin Lists also support basic List.head and List.tail features, as well as pattern matching in the following 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")
```

2.4.3 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()
```

2.4.4 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
**/
type Expr

type Lit(num: Int) extends Expr
type 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
```

2.5 Arithmetic Operators

Pumpkin supports all basic arithmetic operators:

- + Used for addition or String concatonation.
- - Used for subtraction or unary negative.
- / Used for division.
- * Used for multiplication.
- % Used for modulus.

```
1 + 2    // => 3
4 / 2    // => 2
4.0 / 2  // => Type error
3 % 2    // => 1
6 - 2    // => 4
6 - -2   // => 8
```

3 Program Structure

Pumpkin is a compiled functional scripting language. Thus the entry point of the code is the first line of a file, and the return type/value is whatever the last line of the file returns.

3.1 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.
```

3.2 Indentation

Pumpkin does not use curly brackets to delimit scope, thus white space is very important. The indentation must be one tab chatacter, and will be used to delimit code blocks.

```
if (even 2):
    if (even 5):
        print "unreachable ensted code"
    else:
      if (True): print "one line can happen as well" else: print "You always
   need a colon though
    print "Even"
else:
  print "Odd"
```

4 Functions

In Pumpkin all flow-control is done through functions. Pumpkin does not support **for** or **while** loops.
Functions can be defined with the def key word. The types must be specified and the parameters must be in parantheses. When calling the function, parentheses are optional and arguments are comma separated.

Examples:

```
def funcName(parameter: type): returnType => code

// 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
```

Three ways of declaring the boolean "is 2 even?":

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

val x = even(2) // value is determined at compile time
val x = even  2// alternative way to call function
```

Anonymous functions are also allowed for flow control.

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

To pass a function as a parameter use the syntax x:(params -¿ return type).

```
def x(y: Int, z:(int, int -> String)): Bool => True
```

The fucntion x takes two parameters; y, which is an int, and z, which is a function. z takes two int parameters and returns a string. x returns a boolean.

4.1 Function Chains

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:

```
val b: Int = 3;
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 Bool.
4. No more pipes, left-to-right is done, return boolean value.

```
val b: Int = 3;
even <| (b |> (x: Int => x + 1): Int)
```

The above expression gets executed as follows:

1. Evaluate expression even: (x: Int => Bool).
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 => Bool) 'even' function.
7. After evaluating there are no more pipes or operations, return boolean.

```
val b: Int = 3;
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.

4.1.1 Multi-line Piping

Pipes will ignore whitespace. Thus if a line begins with a pipe it will push the return of the line above in. Once a line does not begin with a | > or < | sign it is outside the piped block

```
4
|> (+ 1)
|> even // return false

even 2 // outside pipe
```

4.2 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 order or type of the arguments don't match, 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
```

4.3 Partially Applied Functions

Pumpkin supports partially applied functions. When a function is called without all of the necessary arguments, it returns another function that has the previously passed arguments already set. Use '_' to handle ordering in the case that an argument is left out prior to subsequent argument:

```
def plus(x, y):Int =>
x + y

val plusOne = plus(1)
val three = plusOne 2

def divide(x, y):Int =>
x / y

val divideByThree = divide(_, 3)
val two = divideByThree 6
```

Furthermore, if an operator is wrapped with (), it becomes a function which can be partially applied:

```
(+) 1, 3 // Returns 4
(+)      // Returns function that takes two arguements
(+ 1)    // Returns function that takes one argument
(+ 1) 3  // Returns 4
(+ 1, 3) // Returns type error, this is not allowed
```

Another example: Assume map's signature is (function, iterable) and filter's signature is (function, accumulator, iterable) then:

```
[1, 2, 3, 4] |> filter even |> map (* 2) |> fold (+), 0
// Returns '12'
```

4.4 Native Functions

4.4.1 Print

Printing to the standard output stream is akin to Java, and beigns with the word *print*.

Escape characters include \t for tab, \n for newline, and \\ for backslash. As mentioned prior, string interpolation begins with the hash sign # and the statement must be enclosed in brackets.

4.4.2 List Functions

List.fold applies a given function to each element of a list, first calculating the result on an initial value, then combining the results with subesquent elements in the list, and returning a result.

List.map applies a given function to each element of a list and builds a list with the results returned.

List.filter returns all the elements of a list that satisfy a given predicate. The order of the elements in the input list is preserved.