JFun

Language Specification

A functional language for the Java Virtual Machine

1. Introduction

JFun (pronounced “jay-fun”) is a functional object-oriented, and (strongly) type-safe programming language with immutable variables. Based on Java and F#, but differing in the following ways:

1. JFun runs on the Java Virtual Machine rather than the .NET framework.
2. JFun's syntax more closely resembles that of java but still works in a functional way.

1.1 Hello World

println(“Hello World”)

This statement is the equivalent of Java's System.out.println();

1.2 Program Structure

The key organizational constructs in Jfun are as follows

1. The “set” keword, like “let” in f# but portrays the permanence of the binding better. Set assigns a symbol to a variable or function.
2. All functions return a value with the exception of println
3. Function definition looks like this: set add(x, y)={x+y}
4. Unlike F# white space is irrelevant because of the use of parenthesis and brackets
5. |> is the parameter passing operator and >| is the parameter receiving operator
6. Expressions are delimited by newlines or semicolons

Sample Program

```
set sqr(x){x*x}
set sumofsqrs(x){
       x
       |>Matrix.map(sqr(>|))
       |>Matrix.sum(>|)
}
```

1.3 Types

There are two kinds of types in JFun ++: primitive types and reference types. Variables of primitive types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable.

Variables are immutable unless the “immutable” keyword is used after “set”. Also JFun is type inferred unless explicitly stated like so

set (variable_name : type) = value

1.4 Statements differing from Java and F#

Expression statement	`set public static void main(args : string[])={`
	`  set x = 0`
	`  printfn(x)`
	`  printfn(x+1)`
	`}`

If then	Like Java not f#

Function declaration	`Set add(x,y)={`
	`x+y`
	`}`
	`// note that the function automatically returns the value without an explicit "return call"`

Pipeline operator and parameter receiving operator	`set sumofsqrs(x){`
	`  x`
	`  |>Matrix.map(sqr(>|))`
	`  |>Matrix.sum(>|)`
	`}`

Lambda expression	`//lambda expression to square x`
	`fun(x)->{x*x}`

1.5 Classes

Classes become very different once you introduce immutable data and mutable data into the same class. To add to this problem F# has two styles of class definition, one where the constructor and member variables are defined implicitly and one where you create them explicitly, this hurts readability. Since the implicit style is almost unreadable especially when immutables are involved and also because it does not resemble class definition in any other language than OCaml I have decided to remove that style from the language entirely. Immutable values MUST be set in the constructor as they cannot be set after the object is created. Member variables and functions are declared with the member keyword.

```
//example class right triangle
Class RightTriangle={
      member private mutable height : int
      member private mutable width : int
      member color : int

      new RightTriangle()={height<-0; width<-0; color="none"}
      new RightTriangle(h,w,c)={ height<-h; width<-w; color=c}}

      member getHeight()={height}
      member getWidth()={width}
      member get Color={color}
      member setWidth(x)={width <- x}
      member setWidth(x)={width <- x}
      member area()={(width*height)/2}
      member getHypotenuse () ={
Math.sqrt((height*height)+(width*width))}
 }
```

1.5.1 Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are three possible forms of accessibility. These are summarized in the following table.

Accessibility	Meaning
public	Access not limited
protected	Access limited to this class or classes derived from this class
private	Access limited to this class

1.5.2 Fields

A field is a variable that is associated with a class or with an instance of a class.

```
class globalPosition={
      member latitude : double
      member longitude : double
      member metersFromSeaLevel: float

      new globalPosition(lat,lon,height)={
      latitude=lat
      longitude=lon
      metersFromSeaLevel=height
}
}
```

1.5.3 Methods

A *method* is a member that implements a computation or action that can be performed by an object or class.
The ***signature*** of a method must be unique in the class in which the method is declared.

1.5.3.1 Constructors

JFun supports both instance and static constructors. An *instance constructor* is a member that implements the
actions required to initialize an instance of a class. A *static constructor* is a member that implements the actions
required to initialize a class itself when it is first loaded.

1.5.3.2 Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have *accessors* that specify the statements to be executed when their values are read or written.

1.5.3.3 Events

An *event* is a member that enables a class or object to provide notifications. Clients react to events through *event handlers*. Event handlers in JFun will use Java's existing swing libraries since they will both run on the JVM.
The following is an example of a beeper program from
http://java.sun.com/docs/books/tutorialJWS/uiswing/events/ex6/Beeper.jnlp
That I have re-written in JFun

```
import java.awt.*
import javax.swing.JFrame
import javax.swing.JPanel
import javax.swing.JButton
import javax.swing.JComponent
import java.awt.Toolkit
import java.awt.BorderLayout
import java.awt.event.ActionListener
import java.awt.event.ActionEvent

class Beeper extends JPanel implements ActionListener={
    member mutable button : JButton

    new Beeper() {
        super(new BorderLayout())
        button = new JButton("Click Me")
        button.setPreferredSize(new Dimension(200, 80))
        add(button, BorderLayout.CENTER)
        button.addActionListener(this)
    }

    member actionPerformed(e : ActionEvent)={
        Toolkit.getDefaultToolkit().beep()
    }

    //gui
    member static createAndShowGUI()={
        //Create and set up the window.
        set frame = new JFrame("Beeper")
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)

        //Create and set up the content pane.
        set (newContentPane : JComponent) = new Beeper()
        newContentPane.setOpaque(true) //content panes must be opaque
        frame.setContentPane(newContentPane)

        //Display the window.
        frame.pack()
        frame.setVisible(true)
    }

    set public static void main(args : string[])={
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            set run()={ createAndShowGUI() }
        })
    }
}
```

1.6 Arrays and Lists (Matrixes)
JFun will have one unified type called matrix that will replace arrays and lists. Matrixes do not need to have a set size. The variables contained in a matrix, also called the *elements* of the matrix, are all of the same type, and this type is called the *element type* of the matrix. The elements in a matrix are contained in square brackets and semicolon separated. The following example allocates a one-dimensional, a two-dimensional, and a three-dimensional array. As well as a matrix defined by using a looping construct.

```
set matrix1D=[1;2;3;4;5]
set matrix2D=["sun" ; "java"]["Microsoft" ; "fsharp"]
set matrix3D=[1 ; 2][11 ; 12][21 ; 22]
set evens=[for i=1 to 10 -> i ]
```

Elements in a matrix can be accessed by their index like so.

```
printfn matrix1D[2]
>> 3
Printfn matrix2D[0][1]
>> java
```

Under the hood matrixes will probably just be some "syntactic sugar" wrapped around Java's ArrayList

1.7 Strings
Since JFun will run on the JVM and be tied to many of the Java libraries JFun will implement the same strings that Java uses taking all of the good and the bad from them. Although strings are objects they can be declared like primitives, also just like java.

```
set name = "greg"

//is equivalent to
set name = new String("greg")
```

To test for the equivalence of strings however the == operator will be overloaded for Strings in JFun. Rather than returning "true" only when two String variables point to the same object in memory, "==" will now return true when their contents is equal. The String.addresEquals() function will now handle reference comparison.

2. Lexical Structure

2.1 Programs
A JFun *program* consists of one or more *source files*. A source file is an ordered sequence of (probably Unicode) characters. Conceptually speaking, a program is compiled using three steps:
1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2 Grammars

This specification presents the syntax of the Shingle programming language where it differs from Java and F#.

2.2.1 Lexical grammar where different from Java and F#

```
matrix =       [ value_list ] matrix
               [value_list]
value_list =   value; value_list
               value
```

2.2.2 Syntactic ("parse") grammar where different from Java and F#

Function call:
```
function_name(argument_list)
```

Function definition:
```
set function_name(argument_list)={expression}
```

Variable definition:
```
set variable_name = value
```

Reference type declaration
```
set reference_name = new class(argument_list)
```

Pipeline and parameter receiving:
```
value
|> function(>|, argument_list)
or
value
|> function(argument_list, >|)
or
value
|> function(argument_list, >|, argument_list)
```

2.2.3 Grammar notation

The lexical and syntactic grammars are presented using BNF *grammar productions*. Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of nonterminal or terminal symbols. In grammar productions, *non-terminal* symbols are shown in italic type, and terminal symbols are shown in a fixed-width font.
The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. For example, the production:

Function definition:
```
set function_name(argument_list)={
      statement_list
}
```

defines a *function definition* to consist of the token Set, followed by the token function_name, followed by the token "(", followed by an *argument_list*, followed by the token ")", followed by an "=", followed by the tokens "{" and *argument_list* , followed by the "}" token to denote the termination of the *function definition*, defines a *statement-list* to either consist of a *statement* or consist of a *statement-list* followed by a *statement*. In other words, the definition is recursive and specifies that a statement list consists of one or more statements.
When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. For example, the production:

statement-list:
  statement
  statement-list statement

defines a *statement-list* to either consist of a *statement* or consist of a *statement-list* followed by a *statement*. In other words, the definition is recursive and specifies that a statement list consists of one or more statements.
Alternatives are normally listed on separate lines, though in cases where there are many alternatives, the phrase "one of" may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line. For example, the production:
real-type-suffix: one of
```
F f D d M m
```
is shorthand for:
real-type-suffix:
```
F
f
D
d
M
m
```

2.3 Lexical analysis

2.3.1 Line terminators
Line terminators divide the characters of a JFun source file into lines.
new-line:
Carriage return character (U+000D)
Line feed character (U+000A)
Carriage return character (U+000D) followed by line feed character (U+000A)
Next line character (U+0085)
Line separator character (U+2028)
Paragraph separator character (U+2029)

2.3.2 Comments
Java's Two forms of comments are supported: single-line comments and delimited comments. *Single-line comments* start with the characters // and extend to the end of the source line. *Delimited comments* start with the characters /* and end with the characters */. Delimited comments may span multiple lines. Comments do not nest.

2.3.3 White space
White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.
whitespace:
Any character with Unicode class Zs
Horizontal tab character (U+0009)
Vertical tab character (U+000B)
Form feed character (U+000C)

2.4 Tokens
There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.
token:
identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator
Tokens in Jfun that are not in Java or F#:

Keywords:	set	member(technicaly in f# but used in a different way)
Operators:	>|	== (overloaded for string contents comparison)

2.4.1 Keywords different from YYY or ZZZ
A *keyword* is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier.
New keywords: one of
set matrix member (technicaly in f# but used in a different way)
Removed keywords:
let var array list type

3. Basic Concepts

3.1 Application Startup

Application startup occurs when the execution environment calls a designated method, which is referred to as the application's entry point. This entry point method is always named main, and since it is run on the JVM it must have one of the following
signatures:

```
set public static void main(args : String[])={…}
set public static void main(args[] : String)={…}
```

As shown, the entry point cannot return a value as the return type is always void.

3.2 Application termination

Application termination returns control to the execution environment. Since return type of the entry point method is void, reaching the outer-most "}" which terminates that method, or executing a return statement that has no expression, results in a termination status code of 0.

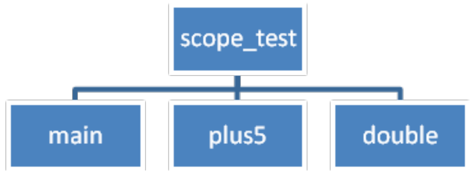
3.3 Scope

JFun uses static scope

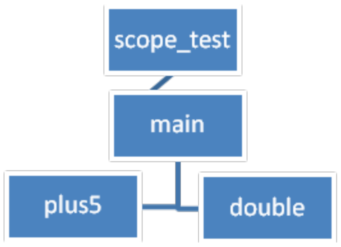
```
class scope_test{
      set public static void main(String[] args)={
            set number = 5

            number
            |>printfn(plus5(>|))
            |>printfn(double(>|))
      }
      set plus5(x)={
            set number=5
            x+number
      }
      set double(x)={
      x*2
      }
}
```

In this example the variable name "number" is used twice but not in the same scope.

Static:



Dynamic:



3.4 Automatic memory management

JFun runs on the Java Virtual Machine it uses Java's Garbage Collector

4. Types

JFun types are divided into two main categories: *Value types* and *Reference types*. Jfun Takes most of its types from Java since it runs on the JVM.

4.1 Value (primative) types

Name	Size	Min	Max	Wrapper
Char	16bit	-128	+127	Char
Byte	8bit	-128	+127	Byte
Short	16bit	-2^{15} *(-32,768)*	$+2^{15}-1$ *(32,767)*	Short
Int	32bit	-2^{31} *(-2,147,483,648)*	$+2^{31}-1$ *(2,147,483,647)*	Int
Long	64bit	-2^{63}	$+2^{63}-1$*)*	Long
Float	32bit	$1.40\text{x}10^{-45}$	$3.40\text{x}10^{+38}$	Float
Double	64bit	$4.39\text{x}10^{-326}$	$1.79\text{x}10^{+308}$	Double
Boolean	1bit	True	flase	Boolean

4.2 Reference types

Objects: An *object* is a *class instance* or an matrix. The reference values are *pointers* to these objects, and a special null reference, which refers to no object. All classes inherit from the class Object. String is an object.

Matrixes: described earlier matrixes are a reference type

5. Types

Variables represent storage locations. Every variable has a type that determines what values can be stored in thevariable. JFun is a strongly is a type-safe language. Variables in JFun are immutable by default. The benefit of immutable variables is that they make concurrency easier to achieve.

5.1 Variable categories

`Non-floating point (byte, short, int, long)`: any number without a decimal point can be stored
Allocated on the stack

```
set x = 3
set mutable y = -6
```

`Floating point (float, double)`: any number with a decimal point can be stored
Allocated on the stack

```
set x = 4.0
set x = (4.45235623465 : double)
```

`char`: any single character can be stored
Allocated on the stack

```
set ch = "g"
```

`boolean`: true or false can be stored
Allocated on the stack

```
set bool = true
```

object: Allocated on the heap

```
set myLocation = new globalPosition(41.721, -73936, 190.5)
```

matrix: Allocated on the heap
set theMatrix = ["Neo", "Morpheus", "Trinity"]

6. Parameter Passing

6.1 Method

In mode: In mode parameter passing is discouraged in JFun although it can be used. But can be used with mutable data types as well as looping structures such as *for* and *while*. In mode parameter passing takes in values and returns nothing.

Out mode: JFun subprograms are encouraged to be out mode parameter passing. With out parameters a sub program takes in a value but does not change it and returns another value. With out mode parameter passing there are no side effects.

In-out mode: Also discouraged in JFun in-out mode parameter passing takes in a value , changes it and then returns a value as well.

6.2 Example

```
class DiceRoll{
      set roll()={
      //generates a random number 1-6
      Math.floor(Math.random()*6)+1
      }

      set rec check(x)={
      //checks to see if all the values in the matrix are true
            Match x with
            | [] -> true
            | [h :: t] when h -> check(t)
            | [h :: t] when !h -> false

      }

      set rec checkoff(oneToSix, count)={
            //counts the number of rolls it takes to get each number at least once
            oneTosix[roll()-1] <- true

            if(check(oneToSix)){
            return count+1
            }
            else
            {
            checkoff(count+1)
            }

      }

      set public static void main(String args[])={
            set mutable =[for i=0 to 6 -> false]
            checkoff(oneToSix, 0)
      }
}
```

Data	main	Local variables:
		Boolean[] oneToSix = [false;false;false;false;false;false]
		Parameters:
		String[] args = null
	roll	Local variables:
		None
		Parameters:
		none
		Return address: checkoff 1
	check	Local variables:
		None
		Parameters:
		boolean[] x
		Return address: checkoff 3
	checkoff	Local variables:
		Parameters:
		Return address: Main line 1
Code	main	`set mutable oneToSix=[for i=0 to 6 -> false]`
		`checkoff(0)`
	roll	`Math.floor(Math.random()*6)+1`
	check	`Match x with`
		`            | [] -> true`
		`            | [h :: t] when h -> check(t)`
		`            | [h :: t] when !h -> false`
	checkoff	
		`oneTosix[roll()-1] <- true`
		`if(check(oneToSix)){`
		`   return count+1`
		`}`
		`else`
		`{`
		`   checkoff(count+1)`
		`}`

7. Conversions

A *conversion* enables an expression to be treated as being of a particluar type.

7.1 Implicit conversions
 There are no implicit conversions in JFun. Functions have specific argument types and only take correct types. However functions can be overloaded to take more than one type.

7.2 Explicit
Explicit conversions can only be performed on mutable variables.

8. Statements

Examples of statements in JFun that are different than in Java and F#

1. set statement
   ```
   set x  = 5
   set sqr(x)={x*x}
   ```

2. Pipeline and parameter receiving
   ```
   set sumofsqrs(x){
         x
         |>Matrix.map(sqr(>|))
         |>Matrix.sum(>|)
   }
   ```

3. Matrix creation
   ```
   set evens=[for i=1 to 10 -> i ]
   ```

4. Lambda expressions
   ```
   fun(x)->{x*x}
   ```

Other statements included in F# or Java that are important but not yet talked about.

1. Match Statement
   ```
   Match x with
   |pattern -> function
   |pattern -> function
   ```

9. Sample Programs

9.1 Fibonacci Number Generator.

```
set rec fibonacci(n)={
    match n with
    | 0 | 1 -> n
    | _ -> fibonacci(n - 1) + fibonacci(n - 2)
}
```

9.2 Even Number Filter and Sorter

```
set rec evenFilter(myMatrix)={
      myMatrix
      |>Matrix.filter( ( >| % 2) == 0)
      |>Matrix.sort(>|)
}
//returns a sorted list of even numbers
```

9.3 Change Maker

```
set rec change(r)={
    match r with
    | :? int when r>=100 -> printfn("amount must be less than 99 cents")
    | :? int when r>=25 -> printfn((r/10)+” quarters”) ; change(r%25)
    | :? int when r>=10 -> printfn((r/10)+” dimes") ; change(r%10)
    | :? int when r>=5 -> printfn((r/5)+” nickles") ; change(r%5)
    | :? int when r>=1 -> printfn(r+” pennies")
    | :? int when r<=0 -> printfn "amount must be 1 cent or more"
    | _ -> printfn "incorrect type"
}
```

9.4 Class Person

```
class Person(){
      member (firstName : String)
      member (lastName : String)
      member mutable (age : int)
      member (sex : String)

      new Person(first, last, _sex)={
            firstName = first
            lastName = last
            sex = _sex
      }

      new Person(first, last, _sex, _age)={
            firstName=first
            lastName=last
            sex = _sex
            age = _age
      }
      member setAge(x)={age <- x}
      member getAge()={age}
}
```

10. Conclusion

JFun is not necessarily an improvement over F# or Java, it is just a different way of looking at functional programming. Creating JFun was an interesting exercise in taking functional principles and using them along with what I know about Java and the JVM to create a hybrid of both. A concurrent language like JFun would actually be very useful when used together with Java because it would excel in concurrency.

I decided to use Javas parentheses and curly brackets. I personally like the way java code looks and I don't care what anyone else thinks. Its my language after all isn't it. Also I got rid of the semicolons. I chose to change a few things from f# some for clarity and some for functionality. Let became set because I personally think that set sounds more permanent than let and variables are encouraged to use immutability. Objects were a mess in f# with multiple ways to set them up including an implicit constructor made in the class declaration. It was disgusting. I chose to throw away that style, as well as others, and adopt one that was more like Javas class definition that almost anyone who has programed with an OO language should be familiar with. One other aspect of class definition I changed was making member the member variable and function initializer. In F# "var" initializes member variables and "member" creates the functions. I chose to just use member for everything. I think my changes make the code much more readable and just as writable as before.

Matrix's are probably my most unique idea. Since processor time and memory are cheep and my time is variable I chose do do away with F#'s lists sequences and arrays, because theyre all basically the same thing but dont all work with the same functions, and Javas arrays and ArrayLists, because array's arent dynamic and the ArrayList syntax is ugy.

The parameter receiving operator was put in because its hard to pipeline into functions that take more than one argument. In f# you would get around it by writing lambda expressions.

In the end JFun is a functional language that runs on the JVM that looks a lot more like and imperative language than any other functional language I can think of. Maybe if people were more comfortable with the way functional languages looked more people would use them.