CMPT383 Assignment 2: Scala

Justin Liu - jzl1 - 301070053

July 20, 2020

Introduction

Scala, a contraction from "scalable language", is a multi-paradigm programming language providing support for both object-oriented and functional programming design. With the capability to design software in the same format as Java and to seamlessly utilize Java libraries, Scala offers tremendous support for the object-oriented paradigm. Groundwork has been laid for functional design by boasting eminent functional features such as currying, immutability, lazy evaluation, pattern matching, first-class functions, higher-order functions, and type inferencing. With Scala offering lending hands in both object-oriented and functional paradigms, Scala is best described as a general-purpose programming language designed for longevity and evolution.

Initialization

Scala code may be executed through the Java Virtual Machine (JVM), converting Scala programs firstly to Java bytecode, and ultimately into code to be run by the JVM. The compiling process is similar to that of Java.

```
scalac "/path/filename.scala" - to compile Java bytecode
scala "/path/filename" - to execute a .class file in the JVM
scala "/path/filename" - to execute a .class file in the JVM
```

Alternatively, a programmer may utilize the Scala REPL, a command-line interpreter that is convenient to utilize for testing code without the hassle of formal compilation.

```
Scala - to initialize the interpreter
:load "/path/filename.scala" - to import predefined functions
```

Features and Characteristics

Compiles to Java Bytecode

Unlike classically compiled (to machine code) languages as C and C++, Scala is compiled to Java bytecode to be executed by a Java virtual machine. The allows Scala to be compiled operating system and hardware independent, offering a more consistent level of performance across machines.

Statically Bound

Like Java, function returns have a declared data type, as such the compiler is aware of which operation is called. However, also like Java, Scala offers the "extends" functionality allowing for some degree of late binding to exist.

```
galaxy extends phones {...} - dynamic binding
```

Strong Typed

Data types must be explicitly declared when they are not inferred. Furthermore, correctness type checks at compile time agree with what is stored in memory.

Type Inference

A characteristic prevalent in higher-level languages, type inference in Scala reduces the tediousness of type declarations where type may be ascertained through return values and literals.

```
var inferType = "my type is inferred"
- type does not need to be declared, the type "String" is inferred
```

First-Class and Higher-Order Functions

Though many programming languages initially lacked the functionality of first-class functions, a pattern commonly seen in modern programming languages is the adoption of this feature. Similarly, Scala provides this feature as well by allowing functions to be declared similarly to variables. As functions may be treated as variables, they may also be returned after a function call.

Immutability

Scala offers a distinction between two variable types, "var" and "val". "var" is an explicit declaration that a variable's value is mutable. Conversely, a "val" declaration is a promise that a variable's value may not be altered, and the variable may not be reassigned.

```
input: var mut = 5
input: mut = mut + 1
output: 6

input: val immut = 5
input: immut = immut + 1
output: error: reassignment to val
```

Lazy Evaluation

On the theme of generality, Scala offers the capability of lazy evaluation through the combination of the "lazy" and "val" keywords. With this feature, a programmer may access infinite data structures and realize improved code performance by evaluating only what is strictly needed. Additionally, Scala makes a distinction between & and &&, the former being the strict version of the AND operator and the latter being the lazy.

```
def ifStrict() : Boolean = { println(Ï'm strict)¨; return false }
def testLazy(){
if(false && ifStrict()){println(¨this branch never used)¨}
else { println(Ïf not strict, I'm lazy)¨}}


input (using && in testLazy, the lazy AND): testLazy()
output: If not strict, I'm lazy
input (using & in testLazy, the strict AND): testLazy()
output: I'm strict If not strict, I'm lazy
```

Pattern Matching

Though many languages adopt some equivalent logical structure offered by the switch/case statements of C and C++, Scala offers a pattern matching tool in theirs. One of the cases possible is the "other" and "_" case, with a similar functionality to Haskell's "otherwise" and "_" for pattern matching.

```
def patternMatch(num: Int) : String = {
num match {
case 1 =¿
"case 1"
case _ =¿
"other cases"
} }

input: patternMatch(1)
output: "case 1"
input: patternMatch(15)
output: "other cases"
```

Currying and Partial Functions

Currying is the transformation of a function that takes multiple arguments, into a function that takes a single argument and returns another function that thereafter may accept further arguments. Using Scala's currying syntax, an intermediate function may be generated by applying and returning a partial function.

```
def curryAdd(a: Int)(b: Int)(c: Int) : Int = { return (a+b+c) }

input: var curryAdder = curryAdd(1)(1)(_)
output: Int =¿ Int = ¡function1¿
input: curryAdder(1)
output: 3
```

Callback

As functions may be stored as variables (callbacks), they may also be passed as a function argument to be called later within the function the callback in embedded within.

```
def cbPrint(callback: =¿ Unit) { callback }

input: cbPrint(println("I'm a callback"))
output: I'm a callback
```

Concurrency

Scala offers several tools to promote concurrency in a program. Of the several tools, Runnable, an explicit definition of a single thread like Java. Additionally, Future may be used to execute a function in a separate thread.

Memory Management

As Scala compiles to Java bytecode bound to run on the JVM, it inherits the automatic memory management system handled by the JVM; that is, unused

objects can be released automatically in a transparent manner (Sarnowski, 2020). Similar to many other languages that employ a memory management system, Scala employs a garbage collector to clear unnecessary objects allocated on the heap, removing the concern for memory leaks.

Similarities and Differences: Strengths and Weaknesses

As Scala is written to run in the JVM and boasts much of the functionality of Java, the language that expresses the most commonality to Scala is Java. The most evident similarity to Java is the ability for code to be written syntactically similar, an intentional design of Scala to maximize interoperability. To elaborate, Scala shares common syntax, object-oriented design, and strongly typed. Interoperability may be further seen by the capability for Scala and Java libraries to be imported seamlessly through a single import statement.

With the similarities between Scala and Java, Scala claims many of the advantages Java also does. Scala thrives within the object-oriented paradigm, currently the most adopted programming archetype. In combination with the interoperability with Java and the already established popularity for Java, Scala has a large community and library support base to draw from. Due to compilation into Java bytecode and utilization of the JVM, Scala offers device and operating system independence. Garbage collection is an additional benefit to the use of the JVM, freeing the programmer from concerns of memory management.

Java is often criticized for the lack of functional programming support. Like Haskell, Scala boasts many features commonly associated with the functional programming paradigm, including type inference, first-class and higher-order functions, immutability (allows for pure functions), lazy evaluation, pattern matching, and partial function application. With these additional tools, Scala fulfills the role as a general-purpose programming language.

While the diverse array of tools offered in the Scala language is its primary advantage, it also may be viewed as its disadvantage. With the adaptability of Scala comes the complexity to match it. Due to Scala's ability to delve into the realms of both object-oriented and functional design, deciphering type-information and function purpose is more complex, possibly hindering

code readability. To elaborate, the adaptability of Scala allows for a myriad of coding styles to flourish (e.g. type inferencing allows for omission of details, currying allows for flexible application of a function), which adds a layer of complexity that must be decoded by a reader. The use of compilation for the JVM also applies a performance concern as optimization may be hindered due to a lack of control over memory management, necessary compilation to the intermediate bytecode, and compilation to a non-native language to a host processor.

Conclusion

Programming languages should not be viewed as strictly superior to each other, but as toolboxes wielding a variety of tools with varying degrees of usefulness dependent on the project. Where top tier efficiency and low-level optimizations are key, C and C++ are your likely choices. If strictly object-oriented modular design is the primary concern, there is Java. If pushing the limits and the experimentation of the functional programming paradigm is the goal, it's going to be Haskell.
Where many other programming languages are more transparent as to their purpose, Scala finds itself in a more unique position. Scala's advantages stem from its flexibility, having the toolset for both functional and object-oriented design. Ironically, this may also be viewed as a weakness due to the complexity that results. Therefore, Scala is a powerful option for projects that benefit from an adaptable language by interweaving both programming paradigms. Scala particularly excels from building upon the success of Java and allowing seamless incorporation of Java libraries and existing Java code bases, while combating the rigid nature of the strict object-oriented design.

Bibliography

[1] Sarnowski, D. (2020). JVM Memory Management. How To Find And Prevent Memory Leaks. Accessed July 20 2020, https://scalac.io/jvm-memory-management