Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-3641 - Lenguajes de Programación I
Enero-Marzo 2018

Ensayo - Proyecto: Características del Lenguaje Scala

Erick Flejan 12-11555
Rafael Cisneros 13-11156

25 de marzo de 2018

Características del Lenguaje Scala.

Scala es un lenguaje de programación estáticamente tipado que integra características de lenguajes orientados a objetos y funcionales, diseñado con el fin de ser un lenguaje de propósito general que permite a sus usuarios crear código de una manera concisa y sin preocuparse por el tipado.

Scala corre en la máquina virtual del lenguaje de programación Java, el cual es el lenguaje raíz más significativo de su creación, y puede trabajar con cualquiera de sus librerías. Muchas de sus decisiones de diseño, como su flexibilidad sintáctica, o su sistema unificado de tipo, fueron hechas para atender problemas que tenía Java.

Estructuras de Control de Flujo de Scala.

Secuenciación

Scala es un lenguaje de programación estructurado, por lo tanto, el orden y la secuenciación de cada expresión, como se lee, es la manera correcta de escribir un programa en este lenguaje. En Scala no existe una distinción entre un *instrucción* y una *expresión*. Se puede definir la combinación de múltiples expresiones mediante el uso de un bloque el cual es definido con *{}*, en este tipo especial de expresiones, la última expresión es el resultado de la expresión completa, y cualquier elemento definido dentro del bloque solo podrá ser alcanzado dentro de este.

```
val x = 42
println(x) // imprime 42
val y = {
  val z = false
  println(z) // imprime false
  8675309
}
println(y) // imprime 8675309
// Desde aqui se puede acceder
// a x o a y, pero no a z
```

Selección

La selección en Scala es manejada por la expresión *"**if** (condición) bloque **else** bloque"*, en donde el valor de cada bloque, define al valor final de la expresión de selección, eligiendo una u otra rama dependiendo del valor Booleano de la condición. El *else* en una expresión de este tipo no es necesario, y en caso de que se que la condición no se cumpla, esta expresión devolvería el valor por defecto *()*. Este estatus de expresión permite que una expresión *if-else* pueda ser usada en cualquier lugar en el que se espere un valor, y aún más cuando ambas ramas de la expresión son del mismo tipo.

```
val x = 42
if (x > 42) {println("mayor a 42")}
// Esta expresion no hace nada
val y = if (x < 100) {
        9001.0
} else {
        9000.5
}
// y tendra un tipo Double
// un valor de 9001.0
```

Existe también un sistema de *pattern matching*, que permite una selección múltiple que depende en la estructura del objeto al que se quiere sujetar la condición de entrada. Se explicará más de este método más adelante.

```
val x = 3
val y = x match {
  case 1 => "uno"
  case 2 => "dos"
  case _ => "mucho"
}
// y sera "mucho"
```

Repetición

Existen varias estructuras de control predefinidas en Scala para manejar la repetición. Algunas de ellas son los "loops":

While loops: Se comportan como en otros lenguajes, poseen una condición y una expresión como un cuerpo, la cuál se evalúa una y otra vez hasta que la condición se mantenga verdadera. Su sintaxis es *while (condición) expresión*

Do-while loops: Funcionan como los *while loops* pero evaluá la expresión del cuerpo antes de comprobar por primera vez la condición. Su sintaxis es como *do expresión while (condición).*

Estas 2 construcciones son llamadas "loops", y no expresiones ya que no regresan un valor distinto al valor único de tipo *Unit*, el cual no es más que un valor por defecto para expresiones sin valores de retorno "interesantes", pero , o lo que se considerarían en otros lenguajes, una instrucción.

```
...
while (n > 0) {
  r = r * n
  n -= 1
}

do {
  println(r)
  r -= 1
}
while (r != 0)
```

Otras expresiones de control de repeticiones son las expresiones "for". Esta expresión devuelve una colecciona partir de otra entrante. Estas colecciones son, en general, objetos que contiene contiene una cantidad, definida o no, de valores u objetos del mismo tipo, como por ejemplo el objeto *Array* que representa a un arreglo inmutable, o el objeto *Map* que contiene un mapa de llaves y valores. Su sintaxis depende de su utilización, algunas de las funciones de esta expresión son:

Iterar a través de colecciones: Lo más simple e intuitivo que puede hacer una expresión for es iterar a través de una lista de elementos de una colección. La sintaxis usual de esta operación es *"**for** (nombre <- colección) cuerpo"*, en donde la *"nombre <- colección"* es una expresión llamada *generator*, consiste en darle un nombre al objeto de la colección para poder utilizar este dentro de la expresión *cuerpo*. En cada iteración, una nueva variable llamada *nombre* es inicializada con el nombre de un elemento de la colección, hasta que se hayan tomado todos. Existe un tipo llamado *Range* el cuál puede ser usado para definir colecciones de rangos, estos son creados con el operador *to* y 2 valores enteros de la siguiente forma.

```
for (i <- 1 to 4)
  println("Iteration " + i)
//Iteration 1
//Iteration 2
//Iteration 3
//Iteration 4
```

Iteración Anidada: Se pueden añadir más de una clausula <-, esto permitirá al for iterar sobre más de una colección a la vez, sin necesidad de usar una expresión "for" extra.

```
for (i <- 1 to 3;
     j <- 2 to 4)
  print(" " + (i * j))
// 2 3 4 4 6 8 6 9 12
```

Filtrar: Cuando no se quiere iterar sobre una colección completa, se puede agregar una clausula al *generador*, llamada un *filtro*. La sintaxis es como *"**for** (nombre <- colección **if** expresión booleana) expresión"*, esto causará que solo se itere cada vez que se genere un objeto que mantenga a la clausula como un verdadera.

```
for(i <- 1 to 5 if (i % 2 == 0);
    j <- 4 to 9 if (j % 3 != 0))
  print(" " + (i + j))
// 6 7 9 10 8 9 11 12
```

Producir una nueva colección: Mediante la sintaxis *"**for** clausulas **yield** cuerpo"* la expresión "for" tendrá un valor tras su evaluación igual a una colección, devolviendo así una colección del mismo tipo de la colección del generador con los valores regresados por la instrucción *yield* como elementos.

```
val forList = for(i <- 1 to 15 if (i % 2 == 0)) yield i
// forList = Vector(2,4,6,8,10,12,14)
```

Abstracción Procedural

Para dividir el flujo de control de un programa, Scala ofrece las subrutinas familiares en varias versiones. La manera más común de expresar una subrutina es por medio de un método, el cuál es el miembro de un objeto, por medio de la palabra clave *def*. Un ejemplo de un método es:

```
object Lineas {
  def procesarArchivo(nombre: String, ancho: Int) = {
    val fuente = fromFile(nombre)
      for (linea <- fuente.getLines())
        procesarLinea(nombre, ancho, linea)
  }
  def procesarLinea(nombre: String,
      ancho: Int, linea: String): Unit = {
    if (linea.length > ancho)
      println(nombre + ">" + linea.trim)
  }
}
```

Aquí se puede ver la sintaxis de los métodos *procesarArchivo* y *procesarValor*, se define, luego del nombre del método, una serie de parámetros (que

puede ser vacío), seguido, opcionalmente en ciertos casos, del tipo del valor al que evaluará la expresión. Los métodos pueden ser llamados, como se ve en la expresión *linea.length*, con la sintaxis *objeto.método(argumentos)*. Desde un método se pueden alcanzar a todos los métodos que pertenezcan al mismo objeto. Si no se quiere contaminar el espacio de nombres de un programa con métodos auxiliares, se puede usar, como en Java, la palabra clave *private* antes de la definición de una función, sin embarga, Scala posee otra alternativa, la inclusión de funciones locales, creando así funciones anidadas con alcance solo para el ambiente de la función padre.

```
def procesarArchivo(nombre: String, ancho: Int) = {
  def procesarLinea(linea: String) = {
    if (linea.length > ancho)
      println(nombre + ">" + linea.trim)
  }

  val fuente = fromFile(nombre)
  for (linea <- fuente.getLines()) {
    procesarLinea(nombre, ancho, linea)
  }
}
```

Se puede especificar un valor de retorno de un método mediante un *return valor* en cualquier parte del bloque de la función, pero, como en un bloque la última expresión es el valor de la expresión completa, no es necesario, y de hecho, es considerado falla de estilo, hacer esto. Un return solo debería ser usado cuando es realmente necesario romper con el orden de la evaluación de la expresión.

Scala es un lenguaje funcional gracias a que se puede definir *funciones de primer tipo*, las cuales pueden ser pasadas como parámetro y devueltas como una función, ya que estas se consideran un valor de la misma manera que un número entero o una cadena de caracteres. De esto se hablará en un punto más adelante, pero se debería saber que los métodos pueden ser coercionados a funciones de primer tipo.

Manejo de Excepciones

El manejo de excepciones en Scala funciona de la misma manera que funciona en Java. Cuando se lanza una excepción, esta tiene un tipo especifico, la computación actual es detenida y se busca por un manejador de excepciones que pueda aceptar esta excepción. Una excepción tiene un tipo Nothing, si una rama de una instrucción de selección lanza una excepción, las otras rama serán las que definan el tipo de la expresión final, debido a que el tipo Nothing artificialmente extiende de todos los tipos.

La sintaxis de captura de excepciones es modelada a partir de el sistema de *pattern matching*. La instrucción *try { serie de expresiones } catch { casos de pattern matching }* pemite identificar la excepción que sucedió en el bloque *try* en el bloque *catch* para ser manejada correctamente, también se puede especificar un bloque *finally {...}* el cual es ejecutado haya o no haya ocurrido una excepción.

```
val url = new URL("http://ldc.usb.ve/usb.gif").openStream()
try {
  process(url)
}
catch {
  case _: MalformedURLException => println(s"Bad URL: $url")
  case ex: IOException => ex.printStackTrace()
}
finally {
  url.close()
}
```

Recursión

Scala permite métodos y funciones recursivas, lo único que se debe tomar en cuanta al construir una función recursiva es que en esta se debe especificar siempre el tipo de retorno, ya que Scala no puede inferir el tipo de esta función cuando se retornan llamadas a si mismas, o se usan como componentes del mismo valor de retorno.

Una función recursiva de cola simple puede ser optimizada por la JVM automáticamente, sin embargo, esta optimización puede ser bloqueada por razones no obvias. Si se necesita asegurar que se remueve la recursion de un función recursiva de cola, se debería agregar la anotación *@tailrec* y asegurarse que la función no será cambiada por una extensión de la clase en la que se encuentra.

Concurrencia

La concurrencia en Scala esta construida encima del modelo de concurrencia de Java; haciendo uso de hilos mediante los objetos *Thread* y las interfaces *Runnable* y *Callable.*

Esta aproximación tradicional está por lo general llena de efectos secundarios, y por lo tanto erores, debido a la mutación de datos compartidos entre hilos. Scala tiene una alternativa haciendo uso del objeto *Future*, para la organización de computo concurrente sin efectos secundarios. Un objeto de tipo *Future* dará un resultado en algún punto del futuro, se usa principalmente para correr tareas que consumen mucho tiempo y el programa podría estar haciendo algo más útil mientras tanto. *Future {...}* creará un hilo que se encargará de correr el bloque proveido, y, luego de su ejecución, dará como resultado la evaluación del bloque completo cuando haya un éxito o la una excepción en caso contrario. *Future* no es la única herramienta de concurrencia que posee Scala, pero es una de las más importantes y fáciles de usar.

```
...

class RunTest extends Runnable {
  def run() = {
    Thread.sleep(10000)
    println(s"Futuro usando hilo comun ${LocalTime.now}")
  }
}

println(s"Presente: ${LocalTime.now}")
val thread = new Thread(new RunTest) //usando hilo de Java
thread.start

val f = Future {
  Thread.sleep(10000)
  println(s"Futuro usando Future at ${LocalTime.now}")
  42
}

// f sera un valor Future(<not completed>) hasta
// que el bloque sea evaluado, entonces sera
// f = Future(Success(42))
```

Scala como lenguaje orientado a objetos.

Scala es un lenguaje puramente orientado a objetos, es decir, todo valor es un objeto y toda operación sobre estos valores es una llamada a un método. Para conseguir esto, Scala posee un sistema de tipos unificado, todo tipo, u objeto, hereda de un súper-objeto común, incluidos los tipos que solían ser primitivos en Java por defecto, como *Int* o *Boole*an, los cuales son representados por la clase Any Val, por lo tanto estos tipos soportan y poseen métodos definidos, y, por consiguiente, los operadores comunes de estos tipos, como los operadores + y *&&*, son métodos que las instancias de *Int* y *Boolean* pueden usar. *Any* es el súper-tipo común de todos los objetos, este provee ciertos métodos como lo son *equals* o *toString*, este tiene 2 subclases directas llamadas *Any Val,* la cual representa los llamados tipos de valor (*Double*, *Float*, *Long*, *Int*, *Short*, *Byte*, *Char* y *Unit*), en donde se encontrarían los

valores primitivos de Java y *AnyRef* los cuales representan a los llamados tipos de referencia, los cuales serían equivalentes a un *Object* de Java.

Los tipos y el comportamiento de cada objeto son especificados y descritos por las *Clases*, las cuales pueden contener métodos, valores, variables, objetos o incluso otras clases, llamados *miembros*, los cuales son los principales actores a la hora de representar el estado y las operaciones de las instancias de esta clase. Las clases son definidas mediante la palabra clave *class* y las nuevas instancias de una clase mediante *new*. Las clases pueden tomar parámetros opcionales que permite, al ser instanciada, crear un objeto con, no solo campos con los nombres de los parámetros, sino métodos de lectura y escritura para estos campos mediante, para un campo llamado *nombre*, los métodos *nombre* para lectura y *nombre_=* y *nombre =* para escritura.

```
class Punto(var x: Int = 0, var y: Int = 0) // Campos x y y
val punto1 = new Point(1)
// Instancia Punto con el campo x = 1
println(punto1.x)  // Imprime 1
punto.y = 2 // Configura el valor de y a 2
```

Scala permite varios niveles encapsulamiento de miembros, por defecto, los miembros de una clase son públicos. Para restringir su acceso los distintos niveles de encapsulamiento son:

- *private[this] miembro*: Solo puede ser alcanzado por la instancia actual de este objeto, este sería el nivel de encapsulamiento más restrictivo.
- *private miembro*: Puede ser alcanzado por la clase actual o por otras instancias de la misma clase, equivalente a *private* de Java.
- *protected miembro*: Igual que *private* pero también puede ser alcanzado por sus clases.
- *private[package] miembro*: Permite al miembro ser alcanzado por métodos encontrados en el paquete *package* actual.

Las clases de Scala son extendibles mediante un sistema de herencia, el mecanismo de subclases hacen posible especializar a una clases, heredando todos los miembros de una superclase y definiendo miembros adicionales, esto se hace usando el palabra clave *extends súper-clase* luego de la definición de una

clase. Una subclase define a un subtipo, es decir, donde se necesita un objeto del tipo de una súper-clases se puede usar un objeto del tipo de una subclase.

En Scala también existen interfaces, llamadas *traits*, los cuales pueden encapsular miembros mediante la definición de métodos o campos que deben ser usados en las clases que extiendan a este, creando así una especie de plantilla para métodos futuros que extiendan de este trait. Scala permite que ciertos miembros de un *trait* estén parcialmente definidos, pero no es posible definir un constructor en un *trait*, por lo tanto este no puede ser instanciado, estos *traits* son extendidos mediante la palabra clave *extend*, pero como no es una clase, luego del primer *extend*, una clase puede heredar de más de un solo trait, agregando la palabra clave *with* seguido del nombre del *trait*, mediante un método mix-in, permitiendo así agregar funcionalidad a una clase, no solo a partir de una superclase sino también de cuantos *traits* se quieran. La instancia de una clase puede ser extendida mediante *with* para aplicar en ella un mix-in único para la clase, logrando así tener un sistema de programación orientada a roles dentro de Scala. Los *traits* además permiten modificar métodos de una clase, lo que permite que estos sean "empilables" en el sentido de que, si una clase extiende de 2 *traits* que modifican al mismo *miembro*, ambos modificarán al método uno seguido de otro.

No existe herencia múltiple propiamente dicha en Scala, esta puede ser simulada mediante la aplicación de mix-ins para la extensión de clases declaradas, sin embargo, esto sigue sin ser herencia múltiple completa, ya que siguen ocurriendo problemas a la hora de heredar de clases con métodos o valores con el mismo nombre. Si se quiere extender a una clase con miembros parcialmente definidos que además tenga un constructor predefinido, esta clase debe ser una clase abstracta. Una clase abstracta es muy parecida a un *trait* en el sentido en que ambas no pueden ser instanciadas, una de las ventajas que tienen estas sobre los *traits* es que las clases abstractas tienen un mayor nivel de portabilidad debido a que no existe un análogo exacto para *traits* en Java.

```
trait Base1 {
  //Lista de Supertipos
  //Any with AnyRef with Base1
  def print() = { println("Base1") }
}

trait A extends Base1 {
  //Lista de Supertipos
  //Any with AnyRef with Base1 with A
  override def print() = {
    println("A")
    super.print()
  // Se delega a un metodo del trait al que extiende
  }
//Implementacion del metodo del trait al que extiende
}

trait B extends Base1 {
  //Lista de Supertipos
  //Any with AnyRef with Base1 with B
  override def print() = {
    println("B")
    super.print()
  // Se delega a un metodo del trait al que extiende
  }
}

class Base2 {
  //Lista de Supertipos
  //Any with AnyRef with Base2
  def print() = { println("Base2") }
  def base2func = { println("Base2 func") }
}

class C extends Base2 with A with B {
  override def print() {
    println("C");
    super.print()
```

```
  }
  // Se crea una cadena de tipos poco intuitiva
  // pero esta tiene sentido debido a lo siguiente

  // Lista de Supertipos, sabemos que
  // Supertipos de Base2
  // Any with AnyRef with Base2 with
  // Supertipos de A
  // Any with AnyRef with Base1 with A with
  // Supertipos de B
  // Any with AnyRef with Base1 with B with
  //
  // tras quitar duplicados, encontramos que
  // Any with AnyRef with Base2 with Base1 with A with B with C
  // es la lista de supertipos de C
  //
  // Cadena de super entonces seria
  // C.super = B
  // B.super = A
  // A.super = Base1
  // Base1.super = Base2
}
val c = new C c.print()
//C
//B
//A
//Base1
```

Existen otro tipo de clases muy importantes en Scala, que se relacionan al pattern matching, una de las estructuras de control más notables y relevantes en los ambientes de programación funcional, *case classes*. Una clase declarada como *case class* es como una clase regular, pero con la diferencia de que estas son usadas mayormente para la creación de objetos inmutables y estas son comparadas y construidas con solo su estructura en mente. Un ejemplo de su comparación por estructura:

```
case class Message(sender: String, recipient: String, body: String)
val message2 = Message("abc@de.com", "xyz@wv.es", "Com va?")
val message3 = Message("abc@de.com", "xyz@wv.es", "Com va?")
val messagesAreTheSame = message2 == message3  // true
```

Scala como lenguaje funcional.

Una función en Scala es cualquier expresión que tome alguna cantidad de valores como parámetro. Scala es funcional en el sentido de que toda función es un valor; esto se logra haciendo uso de su sistema de tipos unificado y el hecho de que toda función es compilada como una clase que, a tipo de corrida es instanciada. La definición de funciones se realiza mediante $=>$, a la izquierda del símbolo $=>$ se definen los parámetros de la función y a la izquierda el valor que esta devolverá. Estas funciones pueden definirse como funciones anónimas *(lambda)* con la sintaxis utilizada, o como funciones con nombre. Las funciones en Scala son tratadas como valores de primera clase. No solo las funciones siempre son valores, cualquier sección de código que en un lenguaje como Java o C se consideraría un *statement* se considera una expresión en Scala, por ejemplo, la asignación retorna un valor de tipo *Unit* el cual es un tipo con un solo valor *( )*.

```
> val moreThanZero = ((x : Int) => x > 0)
    //funcion Int => Boolean
> if (((x : Int) => x > 0)(y)) 7 else 7 + y
   //funcion anonima usada como condicion del if
```

Scala posee un sistema de inferencia de tipos, que permite que las funciones definidas no se deba especificar el tipo que devuelve, esto ayuda a flexibilizar la sintaxis de ciertas funciones como las funciones anónimas. Scala fuerza una distinción entre variable inmutable, definida con *val*, la cual

su valor no puede ser cambiado, y una variable mutable *v*ar, la cual tiene características opuestas. Los objetos también tienen una distinción similar, la cual reside en variantes definidas del mismo objeto, por ejemplo, existen tanto arreglos en los que sus miembros pueden cambiar, llamados *ArrayBuffer*, como arreglos en los que no, llamados *Array*. Esta distinción debe hacerse al momento de crear la variable. Esto beneficia a la creación de funciones puras, comúnmente solicitadas para implementar un verdadero trabajo funcional.

Los valores función son objetos, así que estos pueden ser guardados en una variable, al mismo tiempo, en cualquier momento que se pueda escribir un valor del tipo que da como resultado la función, se puede usar este objeto como un valor de ese tipo mediante una llamada común a una función. Scala provee una forma de omitir información redundante al escribir funciones. Mediante el sistema de inferencia de tipos se puede omitir el tipo de los parámetros de una función. También existe una notación llamada *Placeholder Syntax*, la cual consiste en usar el símbolo _ como un nombre genérico para el parámetro de entrada, esta sintaxis es mayormente usada con operadores.

```
> val someNumbers = Array(-1, 0, 1)
> someNumbers.filter(x => x < 0) // Array(true, true, false)
> someNumbers.filter(_ > 0) // Array(false, false, true)
```

Por defecto, la evaluación de expresiones en Scala es estricta, es decir, apenas son disponibles estas son evaluadas. Sin embargo, una variable puede declararse mediante la palabra clave *lazy*, significando que el código que compone a esta variable no será evaluado hasta el momento en el que la variable sea solicitada.

Scala soporta su propio mecanismo de *pattern matching* para conseguir valores comparados según un patrón. Esta estructura de control es usada mediante *match* en donde el bloque que sigue tiene una serie de patrones y valores de la forma *case patrón => valor*. Este mecanismo no es solo una versión distinta del mecanismo de *switch-case* de Java, ya que los patrones pueden de-construir a un valor en sus partes constituyentes para ser usadas en la evaluación.

Scala posee un tipo especial de definición de *traits* y *clases*, que permite un nivel de privacidad único para sus sub-tipos, en el que estos solo pueden

ser declarados dentro del mismo archivo, esto asegura que todos sus subtipos sean conocidos y no haya ambigüedad en su cantidad. Subtipos que extiendan de estas clases, por lo general son declarados como *case classes*. La extensión de un *sealed trait* o *sealed class*, debido a su seguridad en la cantidad de definiciones y su creación y comparación en base a estructura y no a referencia, los hacen perfecto para su uso en pattern matching de los tipos que extiendan a esta clase o trait.

Scala puede definir funciones dentro de funciones, además de métodos dentro de métodos, las cuales son subrutinas anidadas dentro del alcance de otras, haciéndose invisible a el exterior del alcance de la función que encapsula. Scala permite funciones que tomen a otras funciones como parámetros o retornen funciones como un resultado, estas son llamadas funcionen de orden superior; esto es posible gracias a que las funciones son tratadas como valores de primera clase, y estas funcionen permiten un gran nivel de composición de funciones más complejas. Mediante coerción, el método de un objeto puede ser convertido en una función para ser pasado como parámetro.

Usos y limitaciones del lenguaje.

Aplicaciones web: Se utiliza el framework Play para el desarrollo de aplicaciones web tanto para pcs como para equipos móviles. Scala es perfecto para realizar el back-end y manejo de datos de las aplicaciones, y con el projecto Scala.js, ahora también es posible realizar la interfaz gráfica.

Concurrencia: Con el uso de colecciones inmutables, y la evasión de efectos de borde que provee la programación funcional, se pueden realizar diversas operaciones sobre una colección en diferentes hilos, sin necesidad de usar estrategias de exclusión mutua, ya que ninguno de los hilos podrá hacer modificaciones sobre la colección original. Ademas Scala provee una variedad de librerías para programar concurrencia como Future, Actor, Akka, Clustered Akka y Thread.

Manipulación de Big Data: gracias al enfoque funcional que tiene Scala, es muy utilizado en el modelado de programación para Big Data, donde se toman a una colección inmutable de objetos, se transforman los

datos y se retorna una nueva colección. Principalmente se utiliza la herramienta Spark para este tipo de trabajos.

Entre algunas de las limitaciones que se encuentran al programar en Scala, se puede mencionar la poca presencia en la web de comunidades de programadores de Scala, por lo que es difícil encontrar en la web una solución para un determinado problema surgido, o encontrar ayuda de programadores más experimentados. En la mayoría de los casos de deberá tratar de solucionar el problema sin ayuda.

Otra limitación se encuentra en la migración de una versión de Scala a otra. Las diferentes versiones presentan poca compatibilidad, por lo que cambiar una aplicación de una versión a otra es una tarea que puede llevar mucho tiempo. La reutilización de código entre versiones también es poco factible, así que será necesario reimplementar funcionalidades al cambiar de versión.

Scala es un lenguaje compilado, por lo que ofrece mayor velocidad a la hora de correr programas. Sin embargo, compilar dichos programas puede tomar un tiempo excesivo, lo que ralentiza la producción de los mismos. Para la realización de grandes proyectos es recomendable dividir las tareas en módulos para minimizar el tiempo de compilación, sin embargo se seguirá teniendo un impacto negativo en la productividad.

Comparación con lenguajes similares.

Java: Ya que Scala corre en la JVM, es un lenguaje que presenta muchas similitudes con Java. Sin embargo son lenguajes que también presentan muchas diferencias. Java es mundialmente conocido como un lenguaje donde es necesario escribir mucho, en general Scala permite realizar los mismos programas en muchas menos líneas de codigo. Por ejemplo, la inicialización de una lista seria de la siguiente forma:

```java
//Java:
List<String> list = new ArrayList<String>();
list.add("1");
list.add("2");
list.add("3");

//Scala:
val list = List("1", "2", "3")
```

O recorrer los elementos de una lista, en el siguiente ejemplo, convertir un string en una lista de los enteros que representan cada carácter.

```java
//Java:
List<Integer> ints = new ArrayList<Integer>();
for (String s : list) {
  ints.add(Integer.parseInt(s));
}

//Scala:
val ints = list.map(s => s.toInt)
```

Ambos lenguajes son fuertes y estáticamente tipados, en Java es obligatorio declarar el tipo de cada variable, en Scala la declaración no es obligatoria, el tipo puede inferirse, aunque igual se puede declarar explícitamente el tipo de la variable en cuestión. Java no soporta sobrecarga de operadores, mientras que Scala sí. En Java todo es un objeto, mientras que en Scala todo es una expresión, y tiene un valor.

Haskell: Scala está inspirado en parte por Haskell, por lo que se pueden encontrar ciertas similitudes. En Haskell se tiene evaluación lazy por defecto, en Scala esta función es opcional y debe especificarse. Ambos lenguajes tiene inferencia de tipos, aunque la de Haskell es mucho más poderosa. Las funciones de Haskell no tienen efectos de bordes, Scala implementa esta función con colecciones inmutables, pero puede usarse también las colecciones mutables para obtener efectos de borde. Ambos lenguajes hacen uso de pattern matching. La principal diferencia en que Scala es también un lenguaje con programación orientada a objetos e imperativo, mientras que Haskell no.

Ruby: Ruby es un lenguaje orientado a objetos que comparte ciertas características con Scala. Ambos son reflexivos, por lo que se pueden hacer preguntas como ¿Está definida esta variable? O ¿De qué tipo es? Ambos lenguajes implementa mixins, Scala con los Traits, y Ruby con módulos. Entre sus diferencias se puede encontrar que Scala es un lenguaje compilado, mientras que Ruby es interpretado. Ambos son lenguajes fuertemente tipados, pero mientras Scala realiza el chequeo a tiempo de compilación, Ruby lo hace en tiempo de corrida. Además en Ruby no hay declaraciones de tipo, mientras que en Scala son opcionales.

Proyectos desarrollados en Scala.

Spark: Proyecto desarrollado por la organización Apache Software Foundation, escrito principalmente en Apache, aunque también provee APIs de alto nivel para Python, Java y R. Es un sistema con propósito general de computación en clúster para trabajar con grandes volúmenes de datos (Big Data) de manera rápida. Tiene un motor optimizado para el trabajo con grafos para análisis de datos.

Personalmente pensamos que se usa Scala para este proyecto ya que es fácil de producir código, en similitud con Python, pero es estáticamente tipado y compilado, por lo que llega a ser más rápido que Python. Por correr en la JVM, tiene acceso a las APIs de Java, además de poseer una gran cantidad de APIs propias de Scala para realizar funcionalidades comunes en la programación y que Java requiere de gran cantidad de código para realizar ya que no posee un API tan amplia. También une características de programación orientada a objetos y programación funcional que son muy útiles para la transformación y el procesado de datos, que es la principal función de Spark.

Scala.js: Compilador de Scala a JavaScript, todas las funcionalidades de Scala, incluida inferencia de tipos, pattern matching, herencia de tipo, disponibles para la creación de código JavaScript de forma más segura gracias a que Scala es estáticamente tipado.

El por qué se realiza esta herramienta en Scala, se debe al constante crecimiento del lenguaje Scala como de JavaScript, cada vez más personas usan Scala, y JavaScript es uno de los lenguajes más usados para el desarrollo

web. Realizar las tareas de JavaScript en un lenguaje funcional y orientado a objetos como Scala puede ser un gran atractivo.

Twitter: La interfaz web de Twitter está escrita en Ruby on Rails, pero los mensajes, datos y todo el trabajo pesado de back-end está escrito en Scala. Algunos de los servicios de Twitter que usan Scala son Kestrel, una cola del sistema, Flock para guardar el grafo social como una lista de ids de usuarios capaz de realizar 20000 operaciones por segundo en conjunto con MySQL, Hawkwind para el buscador de personas, Hosebird para el Streaming de los tuits a motores de búsqueda públicos, entre otros servicios privados.

Se usa Scala para realizar el back-end ya que es mucho más rápido que Ruby on Trails para la realización de operaciones, y realizar código es relativamente fácil en comparación con otros lenguajes como Java, o C++.

Proyecto.

Enlace al proyecto:
`https://github.com/rafucisneros/CI-3641-MiniProyectoScala`

Interpretador y Parser de una Calculadora.

Enunciado:

Parser e interprete de una mini-calculadora con un REPL. Es capaz de calcular suma, resta, multiplicación, división, módulo y potenciación de números decimales, además de tener una tabla de símbolos simples que permita asignar variables.

Los operadores de esta calculadora serán, *+, -, *, /, % y ^* para sus respectivas operaciones matemáticas incluidos el *+* y el *-* unarios y el operador *=* para la asignación de variables. Las variables aceptadas será cualquier palabra en minúsculas, y los números serán enteros o con decimales. Se puede asignar a una variable cualquier expresión excepto por otras asignaciones. El REPL imprimirá el resultado de cualquier expresión escrita en la terminal, incluída la asignación, la cuál dará como resultado el valor asignado. Se cerrará al presionar *Enter* sin tener nada escrito.

Ventajas de Scala para la realización del proyecto.

La ventaja principal que ofrece Scala sobre otros lenguajes es la librería de *parser combinators*, esta permite, con solo conocer como funcionan las regulares y las gramáticas libres de contexto, crear un lenguaje muy facilmente, con un gran nivel de personalización, y con un nivel de eficiencia competente, sin necesidad de instalar herramientas externas.

Otro factor de escogencia son las *case classes* y la cómoda comparación estructural que estas permiten, seguido de la capacidad de selección mediante *pattern matching*. Estos 2 elementos pueden ser abstraidos a un arbol sintáctico muy fácil de evaluar.

Referencias