Functions

M.C. Oscar Vargas Torres

1 Functions

Note: We will use ("Scala Language Specification | Scala 2.12" n.d.) as much as possible as an authoritative reference.

1.1 Function types

The following is an example of a simple function:

```
val triple = (n: Int) => n * 3
```

The *type* of this function literal is: `Int => Int`. We could have used the *type annotation* to define triple:

```
val triple: Int => Int = n => n * 3
```

That can be read like:

> `triple` is a function that accepts an `Int` parameter and returns an `Int`

In general ("Function Types" n.d.), to declare a **function type**, write

```
(A, B, ...) => C
println("Hello, World!")
```

where

- `A`, `B`, `...` are the types of the input parameters; and
- `C` is the type of the result.

1.1.1 Naming conventions

Scala names are case sensitive. You will find naming conventions for constants, values, variables and methods in ("Naming Conventions | Constants, Values, Variables and Methods" n.d.).

Can you tell why the following would *not* be a good name?

```
val Triple = (n: Float) => n * 3.0f
```

According to ("Function Types" n.d.):

- Function types associate to the right, e.g. `S => T => U` is the same as `S => (T => U)`.
- An argument type of the form `=> T` represents a call-by-name parameter of type `T`.

1.1.2 Relationship between methods and functions: eta–expansion

There is a close relationship between methods and functions, by means of the eta–expansion. See ("Expressions | Method Values" n.d.). For example,

```
def f1(n: Int): Int = n * 2

# Placeholder syntax
val f2: Int => Int = f1 _

# eta-expansion: equivalent to placeholder syntax
val f3: Int => Int = n => f1(n)
```

Given this close relationship, you will find methods referred to as "functions", although they are not strictly the same.

1.1.3 Return value of a block

The last expression of a block becomes the value that the function returns. For example, the following method returns the value of `r` after the `for` loop (no need for the `return` keyword):

```
def fac(n: Int) = {
  var r = 1
  for (i <- 1 to n)
    r = r * i
  r
}
```

1.1.4 Recursive functions/methods must specify the return type

```
def fac(n: Int): Int =
  if (n <= 0) 1
  else n * fac(n - 1)
```

1.1.5 Varargs syntax

```
def sum(args: Int*): Int = {
  var result = 0
  for (arg <- args)
    result += arg
  result

val s = sum(1 to 5: _*)
```

1.1.6 Procedures have `Unit` return value

```
def box(s : String) { // Look carefully: no =
  // contents elided
}

// (Equivalent) Explicit return type
def box(s : String): Unit = {
  // contents elided
}
```

1.1.7 Scaladoc for `Function1`

Open ("Scaladoc for Standard Library 2.12.7" n.d.) and search documentation for trait `Function1`. You should see documentation for important methods like `apply`, `andThen` and `compose`.

1.1.7.1 `apply`

In the following example (taken from `Function1` scaladoc), the definition of `succ` is a shorthand for the anonymous class definition `anonfun1`:

```
object Main extends App {
   val succ = (x: Int) => x + 1
   val anonfun1 = new Function1[Int, Int] {
     def apply(x: Int): Int = x + 1
   }
   assert(succ(0) == anonfun1(0))
}
```

1.1.7.2 `compose` and `andThen`

It has the following signature

```
def compose[A](g: (A) => T1): (A) => R
```

It models the mathematical function composition. For example, if $f(x) = x + 1$ and $g(x) = 2x$,

$$(f \cdot g)(x) = f(g(x)) = 2x + 1 \tag{1}$$
$$(g \cdot f)(x) = g(f(x)) = 2(x + 1) \tag{2}$$

Using Scala:

```
val f: Int => Int = x => x + 1
val g: Int => Int = x => 2 * x

// f "after" g, or g "then" f
// fg(x) = 2x + 1
val fg: Int => Int = f compose g

// g "after" f, or f "then" g
// gf1(x) = 2(x + 1)
```

```
val gf1: Int => Int = f andThen g
gf1(3)
// equivalently
val gf2: Int => Int = g compose f
gf2(3)
```

1.2 Composition

We have reviewed composition in the mathematical sense. This may seem too theoretical, but is a wonderful tool to get complex solutions from smaller building blocks. We are going to use `atto`, a parsing library that has uses `andThen` to build a new parser from smaller parsers. Spend some time studying (Norris n.d.).

1.2.1 Exercise

- Discuss your understanding on Basic Parsers with others (based on (Norris n.d.)).
- Following (Norris n.d.) tutorial, replicate the example given in your development machine.
- TBD: Build a perser for ...

1.3 Currying

1.4 Generics and parametric polymorphism

1.5 Functions and Dependency Injection

References

"Expressions | Method Values." n.d. Accessed September 27, 2018. https://www.scala-lang.org/files/archive/spec/2.12/06-expressions.html#method-values.

"Function Types." n.d. Accessed September 26, 2018. https://www.scala-lang.org/files/archive/spec/2.12/03-types.html#function-types.

"Naming Conventions | Constants, Values, Variables and Methods." n.d. Accessed September 27, 2018. https://docs.scala-lang.org/style/naming-conventions.html#constants-values-variable-and-methods.

Norris, Rob. n.d. "Atto: Basic Parsers. Atto." Accessed September 27, 2018a. http://tpolecat.github.io/atto/docs/first-steps.html.

———. n.d. "Atto: Parsing Log Entries. Atto." Accessed September 27, 2018b. http://tpolecat.github.io/atto/docs/next-steps.html.

"Scaladoc for Standard Library 2.12.7." n.d. Accessed September 27, 2018. https://www.scala-lang.org/api/current/.

"Scala Language Specification | Scala 2.12." n.d. Accessed September 26, 2018. https://www.scala-lang.org/files/archive/spec/2.12/.