Laboratory: Scala (1)

Summary: We begin our exploration of the Scala programming language.

Prerequisites: Tate, Sections 5.1 and 5.2.

Contents:

- Preparation
- Exercises
  - Exercise 1: Simple Type Checking and Coercion
  - Exercise 2: Integer Overflow
  - Exercise 3: Simple Functions
  - Exercise 4: Type Checking and Coercion, Revisited
  - Exercise 5: Tail Recursion
  - Exercise 6: Overloading
  - Exercise 7: Static or Dynamic
  - Exercise 8: “Lambda” Notation
- For Those with Extra Time
  - Extra 1: Implicit Typing
  - Extra 2: Tic Tac Toe Checker

Preparation

a. Create a directory for the lab.

b. Open a browser window on Tate’s examples, in case you want to try any of them.

Exercises

Exercise 1: Simple Type Checking and Coercion

Tate tells us that Scala is strongly typed, but that it does type coercion when appropriate. We see type coercion in a variety of contexts: Most typically, values are coerced when used with an operator that expects a different type; however, values can also be coerced during assignment.

Let’s start with four basic types: String, integer (Int), real (Double), and an object in a class you create (e.g., Person).

a. For the sixteen combinations (String x String, String x Int, String x Double, String x Person, Int x String, ...), determine the effect of adding values of the two types. (There are sixteen combinations because order of parameters may matter.)
b. For the sixteen combinations, determine the effect of assigning values of one type to another.

**Exercise 2: Integer Overflow**

In some languages (e.g., Java), integers are represented with a fixed number of bits. In others (e.g., Scheme), integers can be arbitrarily large. Determine which is the case in Scala.

**Exercise 3: Simple Functions**

The typical way to define a function in Scala is

```scala
def fun(param: type): return-type = expression
```

a. Write and test a function that returns the result of adding 1 to its parameter.

b. Write and test an `id` function that returns its parameter.

**Exercise 4: Type Checking and Coercion, Revisited**

For each of the sixteen pairs you tried in exercise 1, see what happens when you call a function that expects one type with another type.

**Exercise 5: Tail Recursion**

As we know from experience, because functional languages encourage recursive definitions, many functional languages implement tail-recursion removal.

a. Determine if it is possible to overflow Scala’s stack with a non-tail-recursive recursive function. (E.g., you might implement the naive function to sum the numbers from 1 to $n$.)

b. Determine if it is possible to overflow Scala’s stack with a tail-recursive function.

**Exercise 6: Overloading**

a. Determine experimentally whether it is possible to overload a function name in Scala.

b. Tate’s Person example shows that it’s possible to build an inner constructor in addition to the outer constructor. Determine experimentally whether it is possible to have multiple inner constructors in Scala, with the constructor chosen determined by the type of the parameters.

**Exercise 7: Static or Dynamic**

As you may recall, Ruby and Io don’t type-check the bodies of functions until the function is called. Determine when Scala type-checks the bodies of functions, at compile time or run time.
Exercise 8: “Lambda” Notation

While we’ve defined our functions with

```
def fun(param: type): return-type = expression
```

it is possible to write our function definitions in a more functional way, by assigning a lambda expression to a variable. Figure out how to write such definitions.

For Those with Extra Time

If you find yourself with extra time, attempt either or both of these problems.

Extra 1: Implicit Typing

We’ve been told that an advantage of Scala’s type system is that it can implicitly type values in some cases. For example, if you write

```
val x = 2
```

Scala figures out that \( x \) is an integer. Ideally, Scala could carry this further, and determine the types of parameters to a function. See if there are non-trivial situations in which Scala can determine types.

Extra 2: Tic Tac Toe Checker

Tate suggests that you write a program that checks a Tic-Tac-Toe board for winner.