Exercises week 11
Monday 14 November 201

2011-11-14

Goal of the exercises

The goal of this week's exercises is to get acquainted with some core features of the Scala programming language.

Do exercises 11.1, 11.2, 11.3, 11.4, and 11.5. Hand in the solutions.

Do this first

Download Scala from `http://www.scala-lang.org/downloads` and unpack it for your platform (Windows, MacOS, Linux). To run Scala, you must have a Java Virtual Machine installed too.

Exercise 11.1 Recall the very simple expression language shown in the lecture:

```
sealed abstract class Expr
case class CstI(value: Int) extends Expr
case class Prim(op: String, e1: Expr, e2: Expr) extends Expr

def eval0(expr: Expr): Int =
  expr match {
    case CstI(i)          => i
    case Prim(op, e1, e2) =>
      val v1 = eval0(e1)
      val v2 = eval0(e2)
      op match {
        case "+" => v1 + v2
        case "-" => v1 - v2
        case "*" => v1 * v2
      }
  }
```

(i) Extend the evaluator `eval0` with new cases to handle also operators for division (`/`), less than (`<`), greater than (`>`) and logical and (`&`). In the evaluator, use 1 to represent true and 0 to represent false. Try the new evaluator on several examples, for instance `7 * (10 - 6 + 5)` and `7 * (10 < 6 + 5)`.

(ii) Extend the abstract syntax for expressions with a new case class `Var` to represent named variables. Extend the evaluator `eval0` to create a new evaluator `eval1(expr: Expr, env: Map[String,Int]): Int` that can evaluate expressions involving variables. The variable environment `env` is represented by an instance of Scala's immutable Map class.

An environment that maps variable `x` to 27 and `y` to 42 can be written `Map("x" -> 27, "y" -> 42)`.

Exercise 11.2 Write a Scala function `simplify(expr: Expr): Expr` to perform expression simplification on the extended `Expr` type from exercise 11.1. For instance, it should simplify $(x + 0)$ to x, and simplify $(1 + 0)$ to 1. The more ambitious student may want to simplify $(1 + 0) * (x + 0)$ to x. Hint 1: Pattern matching is your friend. Hint 2: Don't forget the case where you cannot simplify anything.

You might consider the following simplifications, plus any others you find useful and correct. The last one below is trickier than the others:

$0 + e$	$\longrightarrow$	e
$e + 0$	$\longrightarrow$	e
$e - 0$	$\longrightarrow$	e
$1 * e$	$\longrightarrow$	e
$e * 1$	$\longrightarrow$	e
$0 * e$	$\longrightarrow$	0
$e * 0$	$\longrightarrow$	0
$e - e$	$\longrightarrow$	0

Exercise 11.3 Extend the evaluator `eval1` from exercise 11.1 to create a new evaluator `eval2(expr: Expr, env: Map[String,Int]): Option[Int]` that returns `Some(r)` if the evaluation succeeds with result `r`, and returns `None` if the evaluation fails.

The evaluation may fail because it divides by zero or uses a variable that is not in the environment `env`.

As in Java, you can check whether `x` is a key in the map `env` by evaluating `env.contains(x)`, or more Scala-ish, `env contains x`.

Hint: Use pattern matching on the results returned by the recursive calls to `eval2`.

Exercise 11.4 In exercise 11.3 you end up with nested pattern matching on the results of `eval2`. Create a new evaluator `eval3(expr: Expr, env: Map[String,Int]): Option[Int]` that works exactly as `eval2` but uses Scala's `for` expressions instead of pattern matching on `None` and `Some`.

Hint 1: When `v1` and `v2` have type `Option[Int]`, then

```
for (i1 <- v1;
     i2 <- v2)
yield i1+i2
```

has the same meaning as the more verbose

```
v1 match {
  case None     => None
  case Some(i1) => {
    v2 match {
      case None     => None
      case Some(i2) => Some(i1+i2)
    }
  }
}
```

If `v1` has value `None`, the `for` expression has value `None`; and if `v1` has value `Some(17)`, then `i1` will be bound to 17, and the next clauses in the `for` expression will be evaluated in a similar manner.

Hint 2: Your solution will probably need three bindings inside the `for` expression.

Exercise 11.5 The interface Comparable[T] has a method `compareTo(T): Int` exactly as in Java — in fact, Comparable[T] is just the usual Java interface of that name.

Write a Scala method to sort an array `arr` of comparable items:

```
def sort[T <: Comparable[T]](arr: Array[T]): Unit = { ... }
```

For simplicity, you may use selection sort, although it is slow: for each index `i` in the array, iterate over the indices `j = i+1...` and swap `arr(i)` and `arr(j)` if the latter is smaller. This brings the smallest remaining item into position `i`.

Hint 1: To iterate over the indices of the array, use a `for` loop, `for (i <- ...) { ... }`. Hint 2: Remember from the lecture that you can generate a sequence of indices using `0 to (arr.length-1)` and similar.

Class String (also from Java) implements Comparable[String], so try to apply your method to an array of strings, which you can define as follows, for instance:

```
val arr = (List("de", "da", "se", "fr", "us", "au", "cn")) toArray
```