Subversive Scala

Due: Monday, September 24, 2012 at 11:59pm Pacific
Complete this assignment on your own — 227 total points.

Overview

In class, we briefly saw Scala's embarrassment of riches for functional and object-oriented programming. We also saw some of Scala's...interesting syntactic design choices. This assignment introduces a few more concepts and asks you to subvert many of these features for your own nefarious purposes.

1. Warmup: Partially Applied and Curried Functions

(17 total points)

In class, we discussed ways of defining and using partially-applied and curried functions. This section contains a brief warmup that asks you to recall what we discussed in class. Execute the following Scala code, observe the results, and answer the questions below. Place your answers in the file `warmup.txt`.

The "Materials" section contains information on how to get the files you need.

```
def sum3(x:Int, y:Int, z:Int) = x + y + z
val partial = sum3 _
val partialApply1 = partial(1, _:Int, 3)
val partialApply2 = partialApply1(3)
def sum3_curried(x:Int)(y:Int)(z:Int) = x + y + z
```

(a) What is the type of `sum3`, as reported by Scala? Give an English description of the type. *(2 points)*

(b) What is the type of `partial`, as reported by Scala? Give an English description of the type. *(2 points)*

(c) What is the type of `partialApply1`? *(1 point)*

(d) What is the type of `partialApply2`? *(1 point)*

(e) Bind the name `v1` to the result of a Scala expression that uses `sum3` and results in a partially applied function that: takes two integers and adds `10` to them. *(3 points)*

(f) Give an expression that calls `v1` with the arguments `1` and `100`. *(2 points)*

(g) What is the type of `sum3_curried`? *(1 point)*

(h) Bind the name `v2` to a Scala expression that uses `sum3_curried` and results in a curried function that: takes an `Int` and results in a function that: takes another `Int` and adds `10` to the sum of these two `Int`s. *(3 points)*

(i) Give an expression that calls `v2` with the arguments `1` and `100`. (2 points)

Observe: Call-by-value and Call-by-name

Normally, when you call a Scala function and pass a value as an argument, Scala evaluates the argument before calling the function. This behavior is known as *call-by-value* (because the function receives the value). An alternate behavior would be for Scala to *not* evaluate an argument when it is passed but instead to allow the function to evaluate the argument as-needed. This behavior is known as *call-by-name* (because the function causes an argument to be evaluated whenever the function body uses the argument's name). The default evaluation strategy is call-by-value; but Scala provides special syntax that allows you to specify that an argument should be passed by-name. To observe the differences between call-by-value and call-by-name, read the code at the top of the next page, predict what it will do, then execute the code to see if the actual output matches your prediction.

There is nothing to submit for this part (that's why this section doesn't have a number), but you'll need this information to complete the assignment.

```
def printThenResult(b: Boolean) = {
    println("You called?")
    b
}
def doAndByValue(b: Boolean)  = false && b
def doAndByName(b: =>Boolean) = false && b
doAndByValue( printThenResult(true) )
doAndByName( printThenResult(true) )
```

2. Subverting Control *(90 total points)*

Imagine that you are the lead developer for a small company. You love Scala, and you've decided to adopt it for your project. Unfortunately, the three programmers who work for you are idiosyncratic, stubborn, and a little old-fashioned. All of them refuse to use `for` expessions, and each of them insists instead on using their own favorite, old-timey control structure, none of which are supported by Scala. If *you* were to print out all the non-negative even numbers smaller than 10, you might write:

```
for (i <- 0 to 9)
   if ( (i % 2) == 0 )
      println(i)
```

Each of your employees, however, would prefer to write the code in their own special way:

Gil Bates *wants to invert the behavior of* `do... while`.

```
var i = 0
loop_until (i > 9) {
   if ( (i % 2) == 0 )
      println(i)
   i += 1
}
```

D. Gnuth *likes* `while` *loops, but wishes that Scala had* `continue`.

```
var i = -1
while_c (i < 9) {
   i += 1
   if ( (i % 2) != 0 )
      continue
   println(i)
}
```

Nikki Pascal *likes the semantics of Gil's construct, but not the syntax.*

```
var i = 0
repeat {
   if ( (i % 2) == 0 )
      println(i)
   i += 1
} until (i > 9)
```

The body of a `loop_until` *statement should execute at least one time. The same is true of a* `repeat_until` *statement.*

`continue` *causes control to jump back to the top of the loop.*

Fortunately, you know that Scala stands for "Scalable Language". After examining each of your employees' looping constructs, you decide that you can make all of them happy, and you won't have to give up your new favorite language.

(a) Modify `LoopUntil.scala` to implement Gil's looping construct.

(b) Modify `WhileContinue.scala` to implement D.'s looping construct.

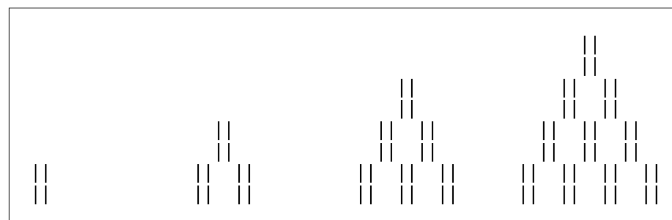
(c) Modify `RepeatUntil.scala` to implement Nikki's looping construct.

You may make the simplifying assumption that the body of each construct has type `Unit`.

3. Block-based Pyramids *(60 total points)*

Using the Block language from class, write a program that displays pyramids of arbitrary size. A pyramid is a block built from two kinds of smaller blocks. A *filled* block is a 2x2 block of pipe characters ('|'). A *blank* block is a 2x2 block of spaces (' '). When displayed, pyramids look like this:

Note: As is often the case, the domain may be partially or ambiguously defined. If anything is not clear, you should ask for clarification.



size: 1, block width: 2, block height: 2 — *size: 2, block width: 6, block height: 4* — *size: 3, block width: 10, block height: 6* — *size: 4, block width: 14, block height: 8*

You may want to modify `Block.scala`*, to make the language more expressive.*

Modify the file `Pyramid.scala`, to implement these block-based pyramids.

4. Time is Fluent *(60 total points)*

In this part of the assignment, you will wrap an existing API in a slightly more fluent interface. The domain is time — in particular, dates on the calendar. You'll use the Java Calendar API as the underlying semantics and layer a syntactic veneer on top of it. The syntax is fairly simple and is defined inductively as follows:

```
n ∈ ℕ
date ::= today
      |  (n days) from (date)
```

For example, the following expressions are valid dates:

```
today
(7 days) from (today)
(3 days) from ((4 days) from (today))
```

All the information you need to get started is in the file `DateLanguage.scala`. Modify this file to implement the date syntax. You can test your implementation using the program in the file `TestDateLanguage.scala`.

Feedback *(participation points)*

Answer the questions in the file `feedback.txt`.

Materials

The materials are available in the Subversion repository. In your working copy, execute:

```
svn copy https://svn.cs.hmc.edu/dsl/fall12/hw/4 hw4
```

Modify the materials to supply your answers and use the instructions below to submit.

Submitting Your Solution

Make sure you're in the `hw4` directory of your working copy and execute:

```
svn commit
```

Committing is like voting in Chicago: you should do it early and often.

Collaboration and Honor Code

I expect you to abide by the Harvey Mudd Honor code. Your solution to this assignment should be produced by you alone. You may discuss concepts at a high level with any student in the class and / or with the course staff. However, you may not copy solutions from anyone, nor may you search for solutions on the Internet.

If you have any questions about what behavior is acceptable, it is your responsibility to come see me before you engage in this behavior. I will be happy to answer any of your questions.