recursive-xml-tree

Work recursively with an XML document tree

Overview

This notebook will show you how to extract text contents from all nodes of an XML document.

Background

We need to introduce two new concepts:

- variables
- pattern matching (which we've seen briefly before)

Variables are values that can change. Normally, we prefer static values (defined with the **val** keyword), but we'll see a situation where a variable is useful.

You create a variable with the **var** keyword. Notice in the example below that you can assign new values to a **var**. If you try to assign a new value you have defined with **val**, you'll get an error.

```scala
val step1 = "String value"
val step2 = "String value, but mutable (ie, can be changed)"

var str = "String value"
require (str == step1)
str = str + ", but mutable (ie, can be changed)"
require (str == step2)
```

// This is not possible: it fails with an error message "reassignment to val"
// step1 = "New value"

Pattern matching is a central feature of Scala. This cell defines a short function that matches on the type of an object. The function **whatIsIt** has one parameter, a scala.xml.Node that could be either an element or a text node. It matches that node for two possible cases: if the node is of type scala.xml.Elem, the function returns one message ("It's an element."); if the node is a scala.xml.Text node, it returns a different message.

```scala
// This is not possible: it fails with an error message "reassignment to val"
// step1 = "New value"
```
def whatIsIt(n: scala.xml.Node) = {
  n match {
    case el : scala.xml.Elem => "It's an element."
    case txt: scala.xml.Text => "It's a text node."
  }
}

import scala.xml._
val tinyXml = "<para>One short paragraph</para>"
val root = XML.loadString(tinyXml)
val expectedAnswer = "It's an element."
val actualAnswer = whatIsIt(root)
require(expectedAnswer == actualAnswer)

Recursion

Like the previous example, this function uses pattern matching to decide is a node is a text node or an XML
element. Here, however, if the node is an element, we use the child method to get a list of all the element’s child
nodes, and apply exactly the same function to them. This is called recursion.

def printTextNodes(n: xml.Node): Unit = {
  n match {
    case txt: xml.Text => { println("TEXT: " + txt.text) }
    case el: xml.Elem => {
      println("(element: " + el.label + ")")
      for (ch <- el.child) {
        printTextNodes(ch)
      }
    }
  }
}
printTextNodes(root)
## A recursive function to collect text

The preceding function walked through every child of an XML element, but didn't return anything useful to us: it just printed out a message. (Note that the *type* of that function was specifically identified as `Unit`, a Scala type that essentially means "this function doesn't return anything").

The following example modifies this slightly to create a recursive function returning a string (and so identifying its type as `String`). It needs *two* parameters: the node to look at, and a string with all text seen so far.

```scala
def collectText(n: xml.Node, s: String): String = {
  var txt = s
  n match {
    case t: xml.Text => {txt = txt + t.text}
    case e: xml.Elem => {
      for (ch <- e.child) {txt += collectText(ch, s)}
    }
  }
txt
}
```

This gives you a good framework to start from. Experiment with more complex XML documents. Think about any tidying up you need to do to account for XML's treatment of white space.