Programming Languages and Paradigms

COMP 302, Winter 2018

Assignment 2

Due date: Wednesday, February 28, 2018
6pm

This assignment focuses on building and validating a parser for the WML language within Scala. Your code must run without error or modification using Scala 2.12.

Note that you must follow the given naming, input, and output requirements precisely. All code should be well-commented, in a professional style, with appropriate variables names, indenting (uses spaces and avoid tabs), etc. **The onus is on you to ensure your code is clear and readable. Marks will be very generously deducted for bad style, lack of clarity, or failing to follow the required instructions.**

Your code should endeavour to follow a pure functional programming style. In particular, and unless specifically stated otherwise, all data types must be *immutable*, and *data may not be modified once assigned or bound*. Note that this means you may not use `var` declarations, `while` or `do`-loops, `ArrayBuffers` or other mutable structures, nor may you reassign `Array` element values after creation.

The goal of this assignment is to be able to successfully parse the WML language described in class, and presented as an explicit grammar in the accompanying *grammar.txt* file.

1. (a) For each of the non-constant tokens in the WML grammar define a regular expression (`Regex` object) that would match it precisely at the start of a string. Bind your regular expressions to `val`'s, named as per the token name, with a capital starting letter and all other letters lowercase. **8**

   Note that "anything" in some of the token descriptions includes whitespace (and newlines too!).

   Answer this question by providing a file `q1a.scala` that contains only the requested val definitions (as something you could cut-and-paste into the REPL), without any outer wrappers.

   (b) Using Scala's `RegexParsers` framework, define a `WMLParser`. Define all tokens in the WML grammar, making use of your regular expressions from the above question. **2**

   Provide a file `q1b.scala` with the class definition.

2. Your parser will need to be able to construct an AST. Define an appropriate class hierarchy for representing your nodes. Your class hierarchy should be rooted at an abstract class, `ASTNode`, and it should be designed to capture all the non-terminals and non-constant/parametric terminals. **5**

   Include `toString` functions for each of your definitions that emit the corresponding name of the terminal token. For non-terminals, the string emitted should start with the terminal name, an opening bracket, and then the entire substructure, all single-space separated, terminated by a closing bracket. Do not print out optional elements that are absent or null. For example, invoking `tostring` on the instantiated root of your tree based on parsing

   ```
   "abc def {{ xxx | bar uuuu }}"
   ```

   should return the string

   ```
   "PROGRAM ( OUTERTEXT INVOKE ( ITEXT ( INNERITEXT ) TARGS ( ITEXT ( INNERITEXT ) ) ) )"
   ```

   Provide a file `q2.scala` with the hierarchy definition.

3. Add grammar rules to your parser, following the grammar specified. The starting non-terminal should be called `program`, and you should generate a tree of `ASTNode` objects.

   You can answer this with a single codebase that parses the entire grammar. You may find it helpful to split up the task, however.

(a) Ensure you can parse basic OUTERTEXT with no template invocations or definitions. **2**

(b) Additionally incorporate the presence of template invocations (ignoring variables and definitions). **6**

(c) Additionally incorporate the potential for variable references (ignoring definitions). **4**

(d) Additionally incorporate template definitions. **8**

For this question you need to define a main (or extend App)—provide a complete scala program that accepts one command line argument which specifies the name of a file that contains a WML program. As output your program should emit to the console the result of invoking `toString` on the root node of your parsed AST.

Provide 4 such program in separate files, `q3a.scala`, `q3b.scala`, `q3c.scala`, `q3d.scala`, each containing a parser that can handle the required input structures, but which you can assume will not be tested on inputs that contain other language elements. Note that these files can be identical, as a solution to part *b* is also a solution to part *a*, and the solution to part *c* is a solution to both *a* and *b*, and the solution to *d* subsumes *a*, *b*, and *c*.

What to hand in

Submit your assignment to *MyCourses*. Note that clock accuracy varies, and late assignments will not be accepted without a medical note: **do not wait until the last minute**. Assignments must be submitted on the due date **before 6pm**.

This assignment is worth 10% of your final grade. **35**