Programming Languages and Paradigms

COMP 302, Winter 2018

Assignment 4

Due date: Wednesday, April 11, 2018
6pm

This assignment focuses on building an evaluator 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.

This assignment focuses on typing and exploring WML as a functional language. You can use your own evaluator or the one provided as a solution to assignment 3. You do not need need to provide your evaluator code, but should assume any WML code you write in this assignment will be tested with the official assignment 3 solution.

1. We can try to encode λ-calculus in WML. To do so, though, we'll need a data structure of some form. You already know that closures can be used to encapsulate data and control access, so that is what we will use.

   Using λ-calculus syntax, we can basic data aggregation with just functions by forming *pairs* of data. The term PAIR $= \lambda xyf.fxy$ constructs a pair of x and y, returning a function which accepts a function that will operate on the pair of elements. We can then access the first part of the pair with the term HEAD $= \lambda p.p(\lambda xy.x)$, which accepts a PAIR construction, and passes in a function that retrieves the first half of the pair. Similarly, the second part of the pair can be retrieved with the term TAIL $= \lambda p.p(\lambda xy.y)$.

   To understand it better, it may be helpful to verify for yourself that *HEAD (PAIR a b)* evaluates to *a*, and *TAIL (PAIR a b)* evaluates to *b*.

   (a) Based on the λ-calculus definitions, define similar *pair*, *head*, and *tail* functions in WML. Verify that `{{head|{{pair|A|B}}}}` == `A`, that `{{tail|{{pair|A|B}}}}` == `B`, and that it continues to work for deeper nesting; e.g., `{{head|{{tail|{{pair|A|{{pair|C|D}}}}}}}}` == `C`. **5**

   (b) Use your pair construction to encode λ-terms in WML. Define constructor functions for building WML-encoded λ-terms as *tagged* pairs—each term is a pair with head (string tag) representing what the tail holds, either a `var`, `app`, or `abs`. The tail then contains the actual data—the actual variable name, or the two terms involved in an application, or the abstraction var and body. **5**

   - Define a function `var` which encodes a base variable v as a pair. Thus the invocation `{{var|x}}` represents the λ-term x.
     To simplify extracting v from your encoding, also define a function `vname` which receives an encoded variable and returns the actual variable. Verify `{{vname|{{var|x}}}}`==`x`.
   - Define a function `app` which encodes an application $M\ N$. Given previously WML-encoded terms `M` and `N`, the invocation `{{app|M|N}}` thus represents the λ-term $(M\ N)$.

To simplify extracting the components, also define functions `app1` and `app2` which return `M` and `N` from an `app` construction respectively.

- Define a function `abs` which encodes an abstraction $\lambda x.M$. Given a base variable x and a previously WML-encoded term `M`, the invocation `{{abs|x|M}}` represents the λ-term $(\lambda x.M)$. To simplify extracting the components, also define functions `param` and `body` which return `x` and `M` from an `abs` construction respectively.

(c) With the above in place, you can define arbitrary λ-terms in WML. But it is not the nicest syntax. Define a WML function `pprint` that receives a WML-encoded λ-term and evaluates into the corresponding, nicely formatted λ-term using standard λ-calculus syntax. Thus, **5**

`{{pprint|{{abs|x|{{app|{{var|x}}|{{var|y}}}}}}}}` == `(λx.(x y))`

You may write "`lambda x`" instead of "λx" if you prefer. Whitespace is not significant (we will stick to single-character variables).

(d) Define a WML `substitute` function that works on your encoding. This function does the bulk of work in β-reduction,[1] substituting a term for all free instances of a given variable within another term. **5**

Your `substitute` function should accept three parameters, M, v, and N, where M and N are WML-encodings of λ-terms, and v is a base variable. It should return a WML-encoding of M$[v \mapsto N]$, replacing all free instances of v in M with N.

Submit a file `q1.wml` with your code, using plain text to indicate each section. Format each function on a new line for readability (blank lines will be ignored in testing).

2. Suppose we have a restricted language, where data may be either a non-pointer, or a pointer (reference) to a non-pointer, or a pointer to a pointer to a non-pointer, etc. We could define a simple formal type-system for our data modeling the levels of pointer indirection with the following inductive type definition,

$$\tau ::= \bullet \mid \mathbf{!}\,\tau$$

where $\bullet$ indicates a non-pointer type (or a pointer to nothing), and $\mathbf{!}\,\tau$ means a pointer to something of type τ. Comparison operations still need a boolean type, so our full type system consists of $\tau \cup \{\text{bool}\}$, although we will only allow actual data to be declared types in τ.

To manipulate pointers, the & and * operators are used for referencing (making a pointer of) and dereferencing (removing one pointer level from) variables, and are applied in a right-associative fashion. For example, if v is a non-pointer (i.e., $v : \bullet$) then we may type the expression $\&\&v : \mathbf{!!}\bullet$, and the expression $*\&\&v : \mathbf{!}\bullet$.

Consider now a program in this language, as below:

```
let x=&y in
   let z=&x in
       let w=*z in
           if (w==x) then
               let r=*&y in r
           else
               let r=**z; in r
```

(a) Come up with a formal set of rules for typing all the code constructs used in the above code. **5**

(b) With the initial assumption that $y : \bullet$, what is the type of the above code? Give a formal proof using your rules to prove your typing is correct. **10**

[1] It does not address inadvertent capture.

Submit a pdf document, q2.pdf, with all fonts embedded, clearly identifying your rules, and showing the full type proof. You may break the proof up into sub-proofs to better fit it on a page, but the structure must be clear.

A handwritten (scanned into pdf) proof is ok, provided your writing is very clear.

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**