Coursework Due: 8pm, 18 April 2019

Instructions Complete parts A and B in the file `Coursework.hs` , and part C in a separate PDF. Submit both on Moodle by Thursday 18 April 8pm. Make sure your Haskell file does not have **syntax errors** or **type errors**; where necessary, comment out partial solutions (and explain what you intended). Use the provided function names. You may use auxiliary functions, and you are encouraged to add your own examples and tests.

Assessment and feedback Your work will be judged primarily on the correctness of your solutions. Secondarily, incorrect or partial solutions (also when commented out) may be given partial marks **if they are well presented.** Marking is part-automated, part-manual. You will receive individual marks, and we will publish an overall feedback document.

Time management The assignments towards the end of this coursework become progressively more difficult. It is possible to spend a large amount of work on the final parts for relatively few marks. If this threatens to become the case, please manage your time carefully and first complete other assignments (for other units).

Plagiarism warning This coursework is an **individual assignment.** Collaboration on this assignment is not permitted: you should not discuss your work with others, and the work you submit should be your own. Disclosing your solutions to others, and using other people's solutions in your work, both constitute plagiarism: see `http://www.bath.ac.uk/quality/documents/QA53.pdf`.

Part A: Lambda-calculus (50%)

In this coursework we will implement the lambda-calculus in Haskell. You are started off with the following, in the file `Coursework.hs`.

- `Var`: a type for variables, as a synonym of `String`.
- `Term`: a data type for lambda-terms. It has three constructors, `Variable`, `Lambda`, and `Apply`, which match the three cases of the definition of lambda-terms:

$$M \quad ::= \quad x \mid \lambda x.M \mid M\,M$$

- `example`: an example lambda-term, $\lambda a.\lambda x.\,(\lambda y.\,a)\,x\,b$.
- `pretty`: a function that renders a `Term` as a lambda-term (but with `\` for λ). Try it:

```
*Main> example
Lambda "a" (Lambda "x" (Apply (Apply (Lambda "y"
  (Variable "a")) (Variable "x")) (Variable "b")))
*Main> putStrLn (pretty example)
\a. \x. (\y. a) x b
```

We will first replace the standard `show` function for `Terms` with `pretty`. Comment out the line `deriving Show` and un-comment the two lines `instance Show Term`. . . .

```
*Main> example
\a. \x. (\y. a) x b
```

Assignment 1 (5%): Complete the function `numeral` which given a number i, returns the corresponding Church numeral N_i as a `Term`. Recall that the Church numerals are:

$$N_0 = \lambda f.\lambda x.\, x \qquad N_1 = \lambda f.\lambda x.\, f\, x \qquad N_2 = \lambda f.\lambda x.\, f\,(f\,x) \qquad \dots$$

You may find the following recursive definition of the numeral N_i helpful.

$$N_i \;=\; \lambda f.\lambda x.\, N_i' \qquad\qquad \begin{array}{ll} N_0' = x & \\ N_i' = f\,(N_{i-1}') & \text{(if } i \neq 0) \end{array}$$

```
*Main> numeral 2
\f. \x. f (f x)
```

Variables

Next, we will build a function that generates a fresh variable. First, we create an infinite supply of variables; then we remove those already in use. We will store used variables as an alphabetically sorted list, with each variable mentioned at most once: we only care **if** variables occur, and not how often. To help with this you are given the `merge` function from the merge sort algorithm in the tutorials.

Assignment 2 (10%):

a) Complete the infinite list `variables`, which contains the variables `"a"` through `"z"`, then repeats these suffixed with 1, `"a1",...,"z1"`, then 2, `"a2",...,"z2"`, etc.

```
*Main> [variables !! i | i <- [0,1,25,26,27,100,3039]]
["a","b","z","a1","b1","w3","x116"]
```

b) Complete the function `filterVariables` which takes two lists of variables and returns the first with all variables from the second list removed from it.

```
*Main> filterVariables ["y","z","a1","a2"] ["y","a1","a3"]
["z","a2"]
```

c) Complete the function `fresh` which given a list of variables, generates a fresh variable not occurring in the list. Use `filterVariables` to remove the given variables from `variables`, then take the first variable in the remaining list.

```
*Main> fresh ["a","b","x"]
"c"
```

d) Complete the function `used` that collects all the variable names used in a `Term`, both as a `Variable` and in a `Lambda` abstraction. Return them in an ordered list (use `merge` to combine two ordered lists into one).

```
*Main> used example
["a","b","x","y"]
*Main> fresh it
"c"
```

Capture-avoiding substitution

In this part we will program capture-avoiding substitution. Recall the renaming operation $M[y/x]$ (M with x renamed to y) from the lectures, slightly paraphrased:

$$\begin{aligned}
z[y/x] &= \begin{cases} y & \text{if } z = x \\ z & \text{otherwise} \end{cases} \\
(\lambda z.M)[y/x] &= \begin{cases} \lambda z.M & \text{if } z = x \\ \lambda z.(M[y/x]) & \text{otherwise} \end{cases} \\
(MN)[y/x] &= (M[y/x])\,(N[y/x])
\end{aligned}$$

The definition of capture-avoiding substitution, similarly paraphrased, is:

$$\begin{aligned}
y[N/x] &= \begin{cases} N & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
(\lambda y.M)[N/x] &= \begin{cases} \lambda y.M & \text{if } y = x \\ \lambda z.(M[z/y][N/x]) & \text{otherwise} \end{cases} \\
& \quad \text{where } z \text{ is **fresh**: not used in } M \text{ or } N, \text{ and } z \neq x \\
(M_1 M_2)[N/x] &= (M_1[N/x])(M_2[N/x])
\end{aligned}$$

Note that both definitions now give a direct template for the corresponding Haskell function.

Assignment 3 (15%):

a) Complete the function `rename x y m` that renames `x` to `y` in the term `m`, i.e. $M[y/x]$.

b) Complete the function `substitute` that implements capture-avoiding substitution, i.e. `substitute x n m` corresponds to $M[N/x]$. Use `fresh` to generate the fresh variable `z` as above; it must not be used in `n` and `m`, and not be `x`.

```
*Main> rename "b" "z" example
\a. \x. (\y. a) x z
*Main> substitute "b" (numeral 0) example
\c. \a. (\a. c) a (\f. \x. x)
```

(In the example, note the renaming of λy to λa, due to the substitution $(\lambda y.\, a)[N_0/b]$.)

Beta-reduction

Now we have all we need to implement beta-reduction. A top-level beta-step is of the form

$$(\lambda x.N)\,M \quad \to_\beta \quad N[M/x]\,.$$

A beta-step can be applied anywhere in a term. This is defined by: if $N_1 \to_\beta N_2$ then

$$\lambda x.N_1 \to_\beta \lambda x.N_2 \qquad N_1\,M \to_\beta N_2\,M \qquad M\,N_1 \to_\beta M\,N_2\,.$$

We will implement a beta-step with the function `beta`. Since a term may have many redexes, or none at all (if it is in normal form), `beta` will return the list of all possible reductions.

Assignment 4 (20%):

a) Complete the function `beta`, which returns the list of all beta-reducts of a term.

```
*Main> Apply example (numeral 1)
(\a. \x. (\y. a) x b) (\f. \x. f x)
*Main> beta it
[\c. (\b. \f. \x. f x) c b,(\a. \x. a b) (\f. \x. f x)]
*Main> it !! 1
(\a. \x. a b) (\f. \x. f x)
*Main> beta it
[\c. (\f. \x. f x) b]
*Main> beta (head it)
[\c. \a. b a]
*Main> beta (head it)
[]
```

Hint: you will need four pattern-matching cases: one to see if the term is a redex, and if not, the three usual cases for `Term` to look further down in the term. In the first case, don't forget to look for further redexes as well. Since `beta` returns a list, you will have to take care with your recursive calls.

b) Complete the function `normalize` which reduces a term to normal form (or continues indefinitely if there isn't one). It should do the following:

- output the current term (use `putStrLn` or `print`),
- apply `beta` to the term (use a `let`-clause),
- if there are no beta-steps, return nothing,
- if there are beta-steps, take the result of the first one and normalize that.

(Your function need not have the same reduction strategy as the example below.)

```
*Main> normalize (Apply (numeral 2) (numeral 2))
(\f. \x. f (f x)) (\f. \x. f (f x))
\a. (\f. \x. f (f x)) ((\f. \x. f (f x)) a)
\a. \b. (\f. \x. f (f x)) a ((\f. \x. f (f x)) a b)
\a. \b. (\b. a (a b)) ((\f. \x. f (f x)) a b)
\a. \b. a (a ((\f. \x. f (f x)) a b))
\a. \b. a (a ((\b. a (a b)) b))
\a. \b. a (a (a (a b)))
```

c) What reduction strategy does your `normalize` function implement? Demonstrate your understanding as follows.

i Make sure that `beta` and `normalize` implement **normal order reduction** (left-most outermost); adjust them if necessary.

ii Copy `beta` and `normalize` to `a_beta` and `a_normalize` and adjust these to implement **applicative order reduction** (left-most innermost). Don't forget to update all the recursive calls.

iii Give a lambda-term `example1` :: `Term` which reduces to normal form in **more** steps with **normal order** than with **applicative order** reduction. Make sure both reductions do actually reach a normal form.

iv Give a lambda-term `example2` :: `Term` which reduces to normal form in **fewer** steps with **normal order** than with **applicative order** reduction. Make sure both reductions do actually reach a normal form.

Part B: The Krivine Abstract Machine (30%)

Our direct implementation of beta-reduction is not very efficient. It repeatedly traverses the term to search for the next redex, to rename a variable, or to carry out a single substitution. An **abstract machine** is a more practical implementation of a given reduction strategy as a state machine. We will build one called the **Krivine Abstract Machine** (KAM).

The KAM relies on two restrictions. First, it assumes **closed** terms, i.e. terms with no free variables, or at least where free variables have different names from bound variables. Second is the reduction strategy, **weak head reduction**:

$$(\lambda x.N)\,M \;\to_{wh}\; N[M/x] \quad \text{and if } N_1 \to_{wh} N_2 \text{ then } \quad N_1\,M \to_{wh} N_2\,M$$

This is a restriction of **normal order reduction** which does not reduce inside an abstraction $\lambda x.N$ or in the argument N of an application $M\,N$. Although it doesn't always find the normal form of a term, weak head reduction is sufficient in practice (since a real-world program doesn't usually return a function $\lambda x.N$). The benefits of weak head reduction are that it is simpler to implement, and with closed terms it cannot produce a reduction where a variable would be captured — so we no longer need to worry about renaming.

First, we avoid searching for each next redex. We can describe weak head reduction by:

$$(\lambda x.N)\,M_1\,M_2\,\ldots\,M_n \;\to_{wh}\; N[M_1/x]\,M_2\,\ldots\,M_n\,.$$

If we organise the applications M_1 through M_n as a **stack** instead, with M_1 at the head, we have direct access to M_1 without having to search at all. We get the following **partial abstract machine** (PAM):

- A **state** is a pair (N, s) where N is a term, and s is a **stack** of terms,

  $$s ::= \star \mid N \cdot s$$

  (where $\star$ is the empty stack).

- The **transitions** are

  $$\begin{array}{rcl} (\lambda x.N\,,\, M \cdot s) & \mapsto & (N[M/x]\,,\; s) \\ (\;NM\,,\; s) & \mapsto & (\;N\,,\, M \cdot s) \end{array}$$

- The **start states** of the machine are those of the form $(N, \star)$.

- The **final states** are those of the form $(\lambda x.N, \star)$ and (x, s).

Assignment 5 (10%):

a) Make the type PState for the states of the PAM such that the example state1, as given, is a PState. (Make sure to capitalize both P and S in PState.)

```
*Main> *Main> state1 :: PState
(\x. \y. x,[Yes,No])
```

b) Complete the functions:
p_start which makes a term N into a start state $(N, \star)$,
p_step which carries out a transition (use your substitute), and
p_final which determines if a state is final.

```
*Main> p_start term1
((\x. \y. x) Yes No,[])
*Main> p_step it
((\x. \y. x) Yes,[No])
*Main> p_step it
(\x. \y. x,[Yes,No])
*Main> p_step it
(\a. Yes,[No])
*Main> p_step it
(Yes,[])
*Main> p_final it
True
*Main> p_final (p_start term1)
False
```

c) Complete the function p_run that carries out a run of the machine. It should initialize a term N as a start state $(N, \star)$, and then loop as follows: output the current state; if it is final, end; otherwise apply a transition step and repeat.

```
*Main> p_run term1
((\x. \y. x) Yes No,[])
((\x. \y. x) Yes,[No])
(\x. \y. x,[Yes,No])
(\a. Yes,[No])
(Yes,[])
```

d) Complete the function p_readback that reads back a Term from a PState by converting the stack back into a sequence of applications. Change p_run so that when it reaches a final state, it prints out the corresponding term (leave the other output and the type of p_run unchanged).

```
*Main> p_run term2
((\b. (\a. \x. (\y. a) x b) Yes) (\z. z) No,[])
((\b. (\a. \x. (\y. a) x b) Yes) (\z. z),[No])
(\b. (\a. \x. (\y. a) x b) Yes,[\z. z,No])
((\c. \a. (\a. c) a (\z. z)) Yes,[No])
(\c. \a. (\a. c) a (\z. z),[Yes,No])
(\b. (\a. Yes) b (\a. a),[No])
((\a. Yes) No (\c. c),[])
((\a. Yes) No,[\c. c])
(\a. Yes,[No,\c. c])
(Yes,[\c. c])
Yes (\c. c)
```

Hint: You can temporarily use `p_readback` for all output of `p_run`, for debugging and to see that it indeed implements beta-reduction.

Completing the machine

In the PAM, we've streamlined the search for the next redex with a clever data structure. To complete the machine, in this part we will similarly use data structures to streamline substitution. The core idea is this: instead of applying substitutions to a term N immediately, we keep them separate, with a pair (N, E) where N is a term and E is a set of substitutions (which we can also represent as pairs). A β-step for such a pair becomes then:

$$((\lambda x.N)\,M\,,\,E) \mapsto (N\,,\,\{\,[M/x]\,\} \cup E)$$

A pair (N, E) is called a **closure**, and the set E of substitutions an **environment**. To change the PAM to the KAM, we replace terms with closures. But if we do that, then a substitution $[N/x]$ must become one $[(N, E)/x]$ as well, and an environment becomes a set of such substitutions. Thus, we define an **environment** E as a stack of triples,

$$E \quad ::= \quad \star \;\mid\; (x, N, E) \cdot E$$

where each triple (x, N, E) represents a substitution of the variable x by a closure (N, E). To retrieve a term from a closure (N, E), we carry out the substitutions. Define $[\![N]\!]_E$:

$$\begin{array}{lcl}
[\![x]\!]_\star & = & x \\
[\![x]\!]_{(y,N,E)\cdot F} & = & \begin{cases} [\![N]\!]_E & \text{if } x = y \\ [\![x]\!]_F & \text{otherwise} \end{cases} \\
[\![\lambda x.N]\!]_E & = & \lambda x.[\![N]\!]_{(x,x,\star)\cdot E} \\
[\![N\,M]\!]_E & = & [\![N]\!]_E\,[\![M]\!]_E
\end{array}$$

(In the case for $[\![\lambda x.N]\!]_E$, a free variable x in N should remain just x. The new environment $(x,x,\star)\cdot E$ does this by making sure x is replaced with itself. Equivalently, we could remove all entries (x,M,F) from E instead.)

The Krivine Abstract Machine is then defined as follows.

- A **state** is a triple (N,E,S) where N is a term, E is an environment — i.e. (N,E) is a closure — and S is a stack of closures,

$$S \quad ::= \quad \star \mid (N,E)\cdot S\ .$$

- The **transitions** are as follows (where $x \neq y$ in the second line)

$$\begin{array}{llllll}
(& x\,, (x,N,F)\cdot E\,, & S) & \mapsto & (N\,, & F\,, & S) \\
(& x\,, (y,N,F)\cdot E\,, & S) & \mapsto & (\,x\,, & E\,, & S) \\
(\lambda x.N\,, & E\,, (M,F)\cdot S) & & \mapsto & (N\,, (x,M,F)\cdot E\,, & & S) \\
(\,NM\,, & E\,, & S) & \mapsto & (N\,, & E\,, (M,E)\cdot S) &
\end{array}$$

- The **start states** of the machine are those of the form $(N,\star,\star)$.

- The **final states** are those of the form $(\lambda x.N,E,\star)$ and $(x,\star,S)$.

Assignment 6 (20%):

a) Make (data)types `Env` for environments and `State` for states. You may use an auxiliary (data)type `Closure` if you wish. Where using datatypes, add `show` functions that display an `Env` as a list of triples and a `State` as a triple (if using `Closure`, display this as a pair, and display the triples in `Env` and `State` as a pair within a pair). Add the following examples as `state2`, `state3`, and `state4`:

$$(\ (\lambda x.x)\,y\ ,\ (y,\lambda z.z,\star)\cdot\star\ ,\ \ \star\ )$$

$$(\ x\,x\ ,\ (x,\lambda x.\,x\,x,\star)\cdot\star\ ,\ \ \star\ )$$

$$(\ \lambda y.x\ ,\ \star,\ (z,\ (z,\lambda a.b,(b,c,\star)\cdot\star)\cdot\star)\cdot\star)$$

```
*Main> state2
((\x. x) y,[("y",\z. z,[])],[])
*Main> state3
(x x,[("x",\x. x x,[])],[])
*Main> state4
(\y. x,[],[(z,[("z",\a. b,[("b",c,[])])])])
```

b) Complete the functions:
`start` which makes a term N into a start state $(N,\star,\star)$,
`step` which carries out a transition, and
`final` which determines if a state is final.

```
*Main> step state2
(\x. x,[("y",\z. z,[])],[(y,[("y",\z. z,[])])])
*Main> step it
(x,[("x",y,[("y",\z. z,[])]),("y",\z. z,[])],[])
*Main> step it
(y,[("y",\z. z,[])],[])
*Main> step it
(\z. z,[],[])
*Main> final it
True
```

c) Implement the KAM as the function `run`. It should initialize a term N as a start state and then loop as follows: output the current state; if it is final, end; otherwise apply a transition step and repeat.

```
*Main> run term1
((\x. \y. x) Yes No,[],[])
((\x. \y. x) Yes,[],[(No,[])])
(\x. \y. x,[],[(Yes,[]),(No,[])])
(\y. x,[("x",Yes,[])],[(No,[])])
(x,[("y",No,[]),("x",Yes,[])],[])
(x,[("x",Yes,[])],[])
(Yes,[],[])
```

d) Complete the function `readback` that reads back a `Term` from a `State`. It should convert the stack back into a sequence of applications while interpreting each closure (N,E) as the term $[\![N]\!]_E$. Change `run` so that when it reaches a final state, it prints out the corresponding term.

```
*Main> readback state2
(\x. x) (\z. z)
*Main> readback state3
(\x. x x) (\x. x x)
*Main> readback state4
(\y. x) (\a. c)
```

```
*Main> run (readback state2)
((\x. x) (\z. z),[],[])
(\x. x,[],[(\z. z,[])])
(x,[("x",\z. z,[])],[])
(\z. z,[],[])
\z. z
```

Part C: Research Question – Continuations (20%)

The **stack** in the Krivine Abstract Machine is a record of what needs to be done after the work being done on the current term is complete. This representation of the "rest of the work" is an example of the concept of **continuation**. This coursework asks you to find out more about this idea, its history, and its applications.

Assignment 7 (10%): Write a short essay (1000–1500 words) on **continuations in programming language theory and practice**. Your essay should address the following points:

- historical background on the discovery of continuations
- a description and explanation of the operation known as "call with current continuation" (call/cc). Where did it originate, what does it do, and how is it used in programming?
- how continuations are used as a programming technique:
  - continuations in modern programming languages
  - continuation passing style (CPS) programming and the CPS transformation
- how continuations are used in the implementation of programming languages (that is, in compilers)

You should make use of the scientific literature on programming to find out about continuations and inform your answer. You will find many Wikipedia entries and blog posts about continuations. These are a good starting point in your search for literature to refer to, but are **not** definitive references. By all means use them to develop your understanding, but aim to seek out scientific research papers, textbook descriptions, and similar definitive documents to support your answer. Marks are allocated not only for your answer but also for the choice of reference sources and the presentation of your bibliography.

Submission and Marking Scheme

Submit your essay as a PDF file named `Coursework_C.pdf` on Moodle, for the assignment labelled **Coursework part C**, by the 18 April 8pm deadline. Preferably, write your essay using appropriate software such as LaTeX or Word. You may write it by hand and scan it to PDF if the result is **clearly legible** and **well presented** (we may refuse poor-quality phone pictures). The 20 marks available for this part are allocated as follows:

- historical background: 2%
- call/cc: 6%
- continuations in programming languages and compilers: 8%
- literature sources and bibliography: 4%