Coursework Due: 8pm, 1 May 2020

Instructions Complete the given assignments in the file `Coursework.hs`, and submit this on Moodle by Friday 1 May 8pm. Make sure your 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.

Plagiarism warning The assessed part of this coursework is an **individual assignment**. Collaboration 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`.

This coursework consists of two parts. In Part A we will implement the lambda-calculus in Haskell, following the definitions of the lectures. This part is **not assessed**, and the solutions are published in `Coursework-Part-A.hs` for you to use. In Part B, which **is assessed**, we will program a more efficient implementation of the lambda-calculus via **supercombinators**—the way that Haskell is implemented at a low level. Since we will build on the code for Part A, you are **strongly encouraged** to attempt this part yourself, without checking the solutions, so that you are familiar with it.

Part A: Lambda-calculus (Not assessed)

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, $\lambda a.\lambda x.\,(\lambda y.\,a\,c)\,x\,b$.
- `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"
  (Apply (Variable "a") (Variable "c")))
  (Variable "x")) (Variable "b")))
*Main> putStrLn (pretty example)
\a. \x. (\y. a c) 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 c) x b
```

Assignment 1: 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:

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

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

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

```
*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:

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","c","x","y"]
*Main> fresh it
"d"
```

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:

$$z[y/x] = \begin{cases} y & \text{if } z = x \\ z & \text{otherwise} \end{cases}$$

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

$$(MN)[y/x] = (M[y/x])\,(N[y/x])$$

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

$$y[N/x] \quad = \quad \begin{cases} N & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

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

where z is **fresh**: not used in M or N, and $z \neq x, y$

$$(M_1 M_2)[N/x] \quad = \quad (M_1[N/x])(M_2[N/x])$$

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

Assignment 3:

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 c) x z
*Main> substitute "b" (numeral 0) example
\d. \a. (\a. d 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

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

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

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

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:

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 c) x b) (\f. \x. f x)
*Main> beta it
[\d. (\b. (\f. \x. f x) c) d b,(\a. \x. a c b) (\f. \x. f x)]
*Main> it !! 1
(\a. \x. a c b) (\f. \x. f x)
*Main> beta it
[\d. (\f. \x. f x) c b]
*Main> beta (head it)
[\d. (\a. c a) b]
*Main> beta (head it)
[\d. c b]
*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:

- apply `beta` to the term (use a `where`-clause),
- if there are no beta-steps, return the input term,
- if there are beta-steps, take the result of the first one and normalize that.

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

c) It is inconvenient that `normalize` does not produce intermediate steps. Adapt it as an IO-function `run` that does 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> run (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)))
```

That concludes Part A. For further testing (and your entertainment), and for use in Part B, your `Coursework.hs` file contains the following further operations on Church numerals, as lambda-terms. (The term `dec` is the **decrement** operator.)

`suc` `add` `mul` `dec` `minus`

Part B: Supercombinator reduction (Assessed)

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. We will build a more efficient implementation using **supercombinators**, as used in the implementation of Haskell.

The idea behind this approach is that we want to compile a lambda-term as a **program** that looks just like a Haskell one. For example, we would like to compile the lambda-term for $1 + (2 \times 3)$ with Church numerals to the following program, and then execute `$main`.

```
$1 f x       = f x
$2 f x       = f (f x)
$3 f x       = f (f (f x))
$add m n f x = m f (n f x)
$mul m n f x = m (n f) x
-------------------------
$main        = $add $1 ($mul $2 $3)
```

A program, in this way, is a set of instructions `$i` $x_1 \ldots x_n$ `= E` where `$i` is a **program variable**, the name of the instruction; x_1 through x_n are its **parameters**; and the **expression** `E` is made up of function applications and variables, where each variable is either one of the parameters x_i or the name of another instruction (or the same instruction, for recursion). A program like this can be compiled into efficient machine code, where every instruction becomes a series of fixed machine operations. In this exercise, we will not go that far: we will only give a naive way of executing a program—which is still much faster than our `normalize` operation from Part A.

The first step of translating a lambda-term into a program is to make it into a **supercombinator**, which is a lambda-term that is structured like a program. A supercombinator S is a term where every subterm that is an abstraction, i.e. of the form $A = \lambda x_1 \ldots \lambda x_n . N$ (that is not directly under another abstraction), has no free variables. Formally, we define a **supercombinator** S and a **supercombinator expression** E_X over a set of variables X as follows:

$$E_X ::= x \in X \mid S \mid E_X \; E_X \qquad\qquad S ::= \lambda x_1 \ldots \lambda x_n . E_{\{x_1,\ldots,x_n\}} \quad (n \geq 1)$$

The fact that a supercombinator S has no free variables means that variable capture is no longer possible. This makes an implementation a lot simpler and more efficient.

First, we will turn a lambda-term into a supercombinator expression by an operation called **lambda-lifting**. Given a term N with free variables x_1 through x_n, we **lift** it to

$$(\lambda x_1 \ldots \lambda x_n . \, N) \, x_1 \; \ldots \; x_n$$

so that $\lambda x_1 \ldots \lambda x_n.\, N$ has no free variables. Note that this operation is using beta-reduction in reverse! Next, we can adjust the definition of lambda-terms N to match that of supercombinators, as follows (where A stands for **abstraction**).

$$N ::= x \mid A \mid NN \qquad\qquad A ::= \lambda x_1 \ldots \lambda x_n.\, N \quad (n \geq 1)$$

Convince yourself that this indeed defines all lambda-terms! We will then transform a lambda-term N with free variables X into a supercombinator expression E_X. To do this, we need to recursively lift all abstractions A in the above definition to supercombinators.

Assignment 5 (30%):

a) Complete the function `free` that returns the free variables of a term in an ordered list with no duplicates.

b) Complete the function `abstractions` which, given a term N and a list of variables $[x_1, \ldots, x_n]$ returns the term $\lambda x_1 \ldots \lambda x_n.\, N$.

c) Complete the function `applications` which, given a term N and a list of terms $[M_1, \ldots, M_n]$ returns the term $N\ M_1\ \ldots\ M_n$.

d) Complete the function `lift` which, given a term N with free variables x_1 through x_n, returns the term $(\lambda x_1 \ldots \lambda x_n.\, N)\ x_1\ \ldots\ x_n$.

e) Complete the function `super` which turns a term into a supercombinator expression. Your function should replace every subterm of the form $\lambda x_1 \ldots \lambda x_n.\, N$ by its lifting, and do the same for subterms inside N. This can be done as follows.

- Recurse on the input term until an abstraction $A = \lambda x_1 \ldots x_n.N$ is found.
- Use an auxiliary function on A to find N (which should **not** be an abstraction), and recursively apply `super` to it.
- Take the result (A with `super` applied to N) and `lift` it.

```
*Main> free example
["b","c"]
*Main> abstractions (Variable "d") it
\b. \c. d
*Main> applications it [Variable "e",Variable "f"]
(\b. \c. d) e f
*Main> lift example
(\b. \c. \a. \x. (\y. a c) x b) b c
*Main> super example
(\b. \c. \a. \x. (\a. \c. \y. a c) a c x b) b c
*Main> normalize it
\g. \e. g c b
```

From supercombinator to program

To transform a supercombinator expression into a program, we follow the inductive definition,

$$E_X ::= x \in X \mid S \mid E_X\ E_X \qquad S ::= \lambda x_1 \ldots \lambda x_n . E_{\{x_1,\ldots,x_n\}} \quad (n \geq 1)$$

and turn every instance of S into an **instruction** `$i` $x_1 \ldots x_n$ = `E` with a fresh program variable `$i`, and the **expression** `E` given by the grammar: `E` $::=$ $x \mid$ `E E` We expect x to be either a program variable `$j` or a parameter x_i, but we cannot enforce that in a data type, so we will allow any variable. We start by building types and data types for expressions `Expr` and instructions `Inst`. You are given the data type `Prog` for programs, and `Show` instances for `Expr` and `Prog`, commented out, plus a list of fresh program variables, `names`.

Assignment 6 (50%):

a) Complete the data type `Expr` for **expressions** with a case `V` for a **variable** and `A` for an **application**. Complete `toTerm` to turn an expression into a lambda-term.

b) Give a type `Inst` for instructions as a triple of a variable (the name), a list of variables (the parameters), and an expression. Un-comment the `Show` instance for `Prog`.

c) Complete `stripAbs` to strip the abstractions from a lambda-term: it should separate $\lambda x_1 \ldots \lambda x_n . N$ (where N is not an abstraction) into a pair $([x_1, \ldots, x_n], N)$.

d) Complete `takeAbs` to give the abstractions A in a lambda-term, following the grammar $N ::= x \mid A \mid N\ N$, as a list.

e) Complete the function `toExpr` which, given a lambda-term N containing the abstractions $A_1, \ldots, A_n$, and given a matching list of fresh program variables $v_1, \ldots, v_n$, returns the expression where each A_i is replaced with v_i in N.

f) Complete the function `toInst` which given a list of fresh names and a pair $(\$\mathtt{i}, N)$ of a name and a term, does the following.

- Separate the input term N into its abstracted variables $x_1 \ldots x_n$ and the remaining body M.
- Take the abstractions $A_1, \ldots, A_m$ from M and replace them with fresh names $v_1, \ldots, v_m$ from the input list to get the expression E.
- Return the triple of: the instruction `$i` $x_1 \ldots x_n$ = E; a list pairing each name v_i with the abstraction A_i it replaces in E; the remaining fresh names.

g) Complete the function `prog` which turns an arbitrary lambda-term into a program. Use the auxiliary function `aux` to recurse on a list of pairs $(\$\mathtt{i}, N)$ of names and lambda-terms (the "to-do list"). Apply `toInst` to each pair to get the next instruction, and

append the list of pairs returned by `toInst` to the "to-do list" for the recursive call to `aux`. To start off `aux`, provide it with a the list of fresh names and the supercombinator for the input term, paired with the variable `$main`.

```
*Main> stripAbs example
(["a","x"],(\y. a c) x b)
*Main> example2
S ((\a. \x. (\y. a c) x b) (\f. \x. x) 0)
*Main> takeAbs it
[\a. \x. (\y. a c) x b,\f. \x. x]
*Main> toExpr ["$1","$2"] example2
S ($1 $2 0)
*Main> let (i,is,_) = toInst names ("$main",example)
*Main> i
("$main",["a","x"],$1 x b)
*Main> is
[("$1",\y. a c)]
*Main> let (i,is,_) = toInst names ("$main",example2)
*Main> i
("$main",[],S ($1 $2 0))
*Main> is
[("$1",\a. \x. (\y. a c) x b),("$2",\f. \x. x)]
*Main> prog example
$main      = $1 b c
$1 b c a x = $2 a c x b
$2 a c y   = a c

*Main> prog example2
$main      = S ($1 b c $2 0)
$1 b c a x = $3 a c x b
$2 f x     = x     -- the order of instructions is not
$3 a c y   = a c   -- important: $2 and $3 could be swapped

*Main> prog example3
$main      = $1 $2 ($3 $4 $5)
$1 m n f x = m f (n f x)
$2 f x     = f x
$3 m n f x = m (n f) x
$4 f x     = f (f x)
$5 f x     = f (f (f x))
```

Running a program

Finally, we will run our programs. The construction we will build is an example of an **abstract machine**: it is defined in mathematics, but it uses only elements that can be directly implemented on a computer, such as addresses, registers, and stacks. Our machine will have a **stack** S of expressions, which is the internal **state**, and a **program** P as we constructed previously. (A real implementation would have direct access to the instructions in a program, but we will keep using a list.) We will write a stack as follows: ε is the empty stack, and $E \cdot S$ is the stack with E as first element and remaining stack S. The machine then transitions from one state to the next, until the stack is empty. The transitions (or steps) are:

$$\begin{array}{lcl} (N\,M) \cdot S & \mapsto & N \cdot M \cdot S \\ \texttt{\$i} \cdot E_1 \cdots E_n \cdot S & \mapsto & (E[E_1/x_1] \ldots [E_n/x_n]) \cdot S \\ & & \text{if} \quad P \quad \text{contains the instruction} \quad \texttt{\$i}\; x_1 \ldots x_n \;=\; E \end{array}$$

That is, if there is an application $N\,M$ at the head of the stack, both parts are put back onto the stack separately. If there is a variable `$i` on the stack, the machine finds the corresponding instruction $\texttt{\$i}\; x_1 \ldots x_n \;=\; E$ in the program P; then takes as many expressions $E_1, \ldots, E_n$ from the stack as there are parameters $x_1 \ldots x_n$; replaces each occurrence of x_i in E with the corresponding expression E_i by a substitution $[E_i/x_i]$; and pushes the result back onto the stack.

Note that there are several things that might go wrong. Firstly, the expression on the top of the stack may be a free variable x, one that is not a program variable. In a real program, which has no free variables, this would not happen—but we would have **constants** instead: numbers, booleans, etc. So we will consider a free variable as representing a constant, or a built-in primitive, and we will simply output the variable name.

Secondly, there might not be sufficient expressions on the stack to process all parameters. This is a real problem, and we will raise an error. This means our machine can only deal with lambda-terms that have sufficient arguments, i.e. those whose normal forms are not an abstraction. Fortunately, we can always add more free variables as arguments to such a term, so that we may reduce it anyway.

Assignment 7 (20%):

a) Complete the function `sub` which applies a list of substitutions $[E_i/x_i]$, given by pairs (x_i, E_i), to an expression.

b) Complete the function `step` which, given a list of instructions and a stack, carries out a transition of the machine as defined above, outputting any free variables.

c) Complete `supernormalize` which given a lambda-term, compiles the term into a program and executes it. The initial stack should contain just the variable `$main`.

```
*Main> let Prog p = prog example2
*Main> let zs = [(x,e) | (x,_,e) <- p]
*Main> zs
[("$main",S ($1 b c $2 0)),("$1",$3 a c x b),("$2",x),("$3",a c)]
*Main> sub zs (V "$main")
S ($1 b c $2 0)
*Main> sub zs it
S ($3 a c x b b c x 0)
*Main> sub zs it
S (a c a c x b b c x 0)

*Main> step p [V "$main"]
[S ($1 b c $2 0)]
*Main> step p it
[S,$1 b c $2 0]
*Main> step p it
S [$1 b c $2 0]        -- output "S"; return [$1 b c $2 0]
*Main> step p it
[$1 b c $2,0]
  ...
[$1,b,c,$2,0]
*Main> step p it
[$3 $2 c 0 b]
  ...
[$3,$2,c,0,b]
*Main> step p it
[$2 c,b]
*Main> step p it
[$2,c,b]
*Main> step p it
[b]
*Main> step p it
b []                   -- output "b"; return []

*Main> supernormalize example
*** Exception: step: insufficient arguments on stack
*Main> supernormalize example2
S b
*Main> supernormalize example4
S S S S S S S 0
```