Coursework **Due: 8pm, Friday 5 May 2023**

Instructions Complete the given assignments in the file `Coursework.hs`, and submit this on Moodle. Make sure your file does not have **syntax errors** or **type errors**; where necessary, comment out partial solutions. 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. Incorrect or partial solutions may be given partial marks if they operate correctly on certain inputs. Marking is part-automated, part-manual. Feedback will be given through an overall document, published on Moodle, and by making solutions available.

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, using other people's solutions in your work, or using an AI-generated solution, all constitute plagiarism: see `http://www.bath.ac.uk/quality/documents/QA53.pdf`.

Combinatory logic

Combinatory logic is a way to express functions without using variables. A **combinator** is a constant with an associated reduction rule, which can be applied to other combinators to give a **combinator expression**. We will use the standard combinators S, K, and I with the following reduction rules: for all expressions x, y, and z

$$\begin{aligned} I\ x &\to x \\ K\ x\ y &\to x \\ S\ x\ y\ z &\to x\ z\ (y\ z)\ . \end{aligned} \tag{1}$$

Note that I is redundant, since the expression $S\ K\ K$ gives the same reduction:

$$S\ K\ K\ x \to K\ x\ (K\ x) \to x\ . \tag{2}$$

As another example, the expression $S\ (K\ K)\ I$ reduces as K:

$$S\ (K\ K)\ I\ x\ y \to K\ K\ x\ (I\ x)\ y \to K\ K\ x\ x\ y \to K\ x\ y \to x\ . \tag{3}$$

As the second step in this example shows, where $I\ x$ reduces to x, reduction may take place anywhere inside an expression.

Combinatory logic (with the SKI-combinators above) is equivalent to the λ-calculus—that is, combinator expressions can be encoded in λ-terms, and λ-terms in combinatory expressions. In this coursework we will implement combinatory logic and both translations. To start off, you are given the implementation of the λ-calculus from the tutorials.

Assignment 1: **(30 marks)**

First, we will implement combinatory logic. We will have **free** variables x, y, z, but not **bound** ones. Combinatory expressions c, d, e are given formally by the following grammar.

$$c ::= I \mid K \mid S \mid c\,c \mid x$$

Here, $c\,c$ is an application; as with the λ-calculus, it associates to the left, so that $c\,d\,e$ means $(c\,d)\,e$. Combinatory expressions do not usually have free variables, but these will be very useful in testing out reductions, and later on in defining the translation from λ-terms to combinators, so we add them from the start.

a) Define combinatory expressions as the data type `Combinator`, in such a way that the examples `example1` and `example2` give the expressions for (2) and (3) above. Use the (binary, infix) constructor `:@` for application, and un-comment the declaration `infixl 5 :@` to make it left-associative.

   Un-comment the two examples, the `Show Combinator` instance, and the parsing function `parse`. The parser is able to read combinators given as strings, including the output of the `show` function.

b) Complete the function `step` to give a list of all the ways to reduce a combinator expression in one step following the rules in (1), similarly to the function `beta` for the implementation of the λ-calculus.

c) Complete the function `run` to fully reduce a combinatory expression, similarly to the function `normalize` for the λ-calculus.

```
*Main> example1
S K K x

*Main> step it
[K x (K x)]

*Main> step (head it)
[x]

*Main> parse "S(SI)(KI)(SIK)I"
S (S I) (K I) (S I K) I

*Main> step it
[S I (S I K) (K I (S I K)) I]
```

```
*Main> step (head it)
[ I (K I (S I K)) (S I K (K I (S I K))) I
, S I (S I K) I I ]

*Main> step (head it)
[ K I (S I K) (S I K (K I (S I K))) I
, I I (S I K (K I (S I K))) I
, I (K I (S I K)) (I (K I (S I K)) (K (K I (S I K)))) I
, I (K I (S I K)) (S I K I) I ]

*Main> run example2
S (K K) I x y
K K x (I x) y
K (I x) y
I x
x
```

Assignment 2: **(10 marks)**

The translation from combinatory logic to λ-calculus is straightforward: the combinators I, K, and S may simply be taken as the following λ-terms.

$$\begin{aligned} I &\mapsto \lambda x.\, x \\ K &\mapsto \lambda x.\, \lambda y.\, x \\ S &\mapsto \lambda x.\, \lambda y.\, \lambda z.\, x\, z\, (y\, z) \end{aligned} \tag{4}$$

a) Complete the function `toLambda` that takes a combinatory expression to a λ-term. Use the interpretation (4) and decide how to translate applications and variables.

```
*Main> toLambda example1
(\x. \y. \z. x z (y z)) (\x. \y. x) (\x. \y. x) x

*Main> toLambda example2
(\x. \y. \z. x z (y z)) ((\x. \y. x) (\x. \y. x)) (\x. x) x y

*Main> normalize it
...
x
```

The translation in the other direction is more interesting. On the surface, it may look completely mysterious how to generate large-scale behaviours with the SKI-combinators.

Things become clearer when we view them as an implementation of **substitution**, for a single argument. Consider the similarities between the left and the right columns:

$$\begin{array}{lcll} x[P/x] & = & P & I\,p \;\to\; p \\ y[P/x]\ (x \neq y) & = & y & K\,y\,p \;\to\; y \\ (M\,N)[P/x] & = & M[P/x]\,N[P/x] & S\,m\,n\,p \;\to\; m\,p\,(n\,p)\,. \end{array}$$

Using this intuition, we can use the SKI-combinators to **abstract** over a variable x in a combinatory expression. That is, we will have "$\lambda x.c$" not as a syntactic construct, but as a transformation of c, that we will write $A(x)(c)$. Then, since we already have variables and application, we can translate λ-terms into combinatory expressions.

The abstraction function A over a variable and a combinatory expression is as follows.

$$\begin{array}{lcll} A(x)(x) & = & I & \\ A(x)(c) & = & K\,c & \text{if } c \text{ is a combinator or variable } y \neq x \\ A(x)(c\,d) & = & S\,(A(x)(c))\,(A(x)(d)) & \end{array} \tag{5}$$

The interpretation $E(M)$ of a λ-term M as a combinatory expression is then as follows.

$$\begin{array}{lcl} E(x) & = & x \\ E(\lambda x.M) & = & A(x)(E(M)) \\ E(M\,N) & = & E(M)\,E(N) \end{array} \tag{6}$$

Assignment 3: **(25 marks)**

a) Implement the abstraction function A given in (5) as the function `abstract`.

b) Implement the interpretation of λ-terms as combinatory expressions E given in (6) as the function `toCombinator`.

```
*Main> toCombinator (Lambda "x" (Variable "x"))
I

*Main> toCombinator (Lambda "x" (Lambda "y" (Variable "x")))
S (K K) I

*Main> toCombinator (Lambda "x" (Lambda "y" (Variable "y")))
K I

*Main> toCombinator (numeral 1)
```

```
S (S (K S) (S (K K) I)) (K I)

*Main> toCombinator (numeral 2)
S (S (K S) (S (K K) I)) (S (S (K S) (S (K K) I)) (K I))

*Main> toLambda S
\x. \y. \z. x z (y z)

*Main> toCombinator it
S (S (K S) (S (S (K S) (S (K K) (K S))) (S (S (K S) (S (S (K S)
(S (K K) (K S))) (S (S (K S) (S (K K) (K K))) (S (K K) I))))
(S (K K) (K I))))) (S (S (K S) (S (S (K S) (S (K K) (K S)))
(S (S (K S) (S (K K) (K K))) (K I)))) (S (K K) (K I)))

*Main> run (it :@ V "x" :@ V "y" :@ V "z")
...
x z (y z)
```

Assignment 4: **(10 marks)**

It is clear that this interpretation produces very large combinatory expressions; in fact, it is exponential. We will demonstrate that here.

a) Complete the functions `sizeL` and `sizeC` to measure the size of λ-terms and combinatory expressions respectively. Each constructor (variable, abstraction, application, combinator) should contribute 1 to the total count.

b) Give a series of λ-terms that grow linearly, but whose interpretations as combinators grow exponentially, as the function `series`. That is, `series n` should be a λ-term for each natural number n, such that `sizeL (series n)` grows linearly in n and `sizeC (toCombinator (series n))` grows exponentially in n. (You may have a different series than the examples below.)

```
*Main> sizeL (numeral 6)
15

*Main> sizeC (example2)
11

*Main> series 0
\a. a
```

```
*Main> series 1
\b. \a. a b

*Main> series 2
\c. \b. \a. a b c

*Main> toCombinator (series 0)
I

*Main> toCombinator (series 1)
S (S (K S) (K I)) (S (K K) I)

*Main> toCombinator (series 2)
S (S (K S) (S (S (K S) (S (K K) (K S))) (S (S (K S) (S (S (K S)
(S (K K) (K S))) (S (K K) (K I)))) (S (S (K S) (S (K K) (K K)))
(K I))))) (S (S (K S) (S (K K) (K K))) (S (K K) I))

*Main> [ sizeL (series i) | i <- [0..9] ]
[2,5,8,11,14,17,20,23,26,29]

*Main> [ sizeC (toCombinator (series i)) | i <- [0..9] ]
[1,19,109,487,1945,7291,26245,91855,314929,1062883]
```

There is a simple way to improve the translation, and by introducing a few further combinators it can be optimized further. The first optimization is to modify the abstraction function A so that it first tests if a variable is free, and translates directly to K if not.

$$A(x)(c) = K\,c \quad \text{(if } x \text{ is not free in } c\text{)}$$

Further optimizations use additional combinators:

$$\begin{aligned}
B\,x\,y\,z &= x\,(y\,z)\\
C\,x\,y\,z &= (x\,z)\,y\\
Q\,u\,x\,y &= u\,x\\
R\,u\,v\,x\,y &= u\,v\,x\\
X\,u\,x\,y\,z &= u\,x\,(y\,z)\\
Y\,u\,x\,y\,z &= u\,(x\,z)\,y\\
Z\,u\,x\,y\,z &= u\,(x\,z)\,(y\,z)
\end{aligned}$$

These work as follows. The combinators B and C are like S: they pass an argument z into an application $x\,y$, but this time only into y or into x (respectively), for use in the case where z is not free in the other branch. For example, if we have an application $x\,y$, then we want to abstract over x as $C\,I\,y$ and over y as $B\,x\,I$. The combinators Q, X, Y, and Z are versions of K, B, C, and Z respectively with one additional argument u, and R is like K with two additional arguments u and v. These are convenient for passing multiple arguments. In our example, we abstract over x in $x\,y$ as $C\,I\,y$, and over y as $B\,x\,I$; to abstract over first x then y we combine both combinators as $Y\,B$, but using Y instead of C. Then Y passes the first argument to x and B the second to y.

$$Y\,B\,I\,I\,a\,b \;\to\; B\,(I\,a)\,I\,b \;\to\; (I\,a)\,(I\,b) \;\to^*\; a\,b$$

Similarly, to abstract first over y and second over x, we get $X\,C\,I\,I$ by using X instead of B. To pass more than two arguments into an application, we can create a stack of combinators, e.g. $Z\,(X\,(R\,C))\,x\,y$ will pass the first argument to both x and y, the second only to y, the third to neither (it will be deleted), and the fourth only to x.

$$Z\,(X\,(R\,C))\,x\,y\,a\,b\,c\,d \;\to^*\; x\,a\,d\,(y\,a\,b)$$

Note that the combinators Q, R, X, Y, Z have different names in the literature—for example, $X = B'$ and $Y = C'$. They have been changed for this coursework so that we can use single symbols for each combinator (so that the parser can stay simple).

Assignment 5: **(25 marks)**

We will optimize the interpretation of λ-terms as combinators. You may use any of the above combinators, and any technique you can find in the literature. For any combinator you add, update the `Show Combinator` instance accordingly, as well as the `parse` function and the `step` function. Make sure that also the `sizeC` function continues to work.

a) Complete `comb` as an optimized interpretation of λ-terms into combinators.

b) Use the function `claim` to give the growth factor of your interpretation, choosing from the options provided by the data type `Complexity`. E.g. to say you have cubic growth, use `claim = Cubic`.

```
*Main> comb (toLambda S)
Y (X S) (Y B I I) (Y B I I)

*Main> run (it :@ V "x" :@ V "y" :@ V "z")
...
x z (y z)

*Main> [ sizeC (comb (series i)) | i <- [0..9] ]
[1,7,15,25,37,51,67,85,105,127]
```

Appendix: when is an interpretation correct?

For an interpretation of λ-terms into combinators, what is to stop someone from simply taking every term to the combinator I and claiming "constant" complexity? Intuitively, this is obviously not correct: it doesn't capture the behaviour of λ-terms in combinators at all. But then, how do we make clear, formally, what an interpretation should achieve, for it to be correct?

Let's start with the other direction: the interpretation of combinators as λ-terms (4), repeated below.

$$I \;\mapsto\; \lambda x.\, x \qquad K \;\mapsto\; \lambda x.\, \lambda y.\, x \qquad S \;\mapsto\; \lambda x.\, \lambda y.\, \lambda z.\, x\, z\, (y\, z)$$

It is clear that if $c \mapsto M$ then M behaves "the same way" as c. To express that formally, we say that the interpretation $\mapsto$ of combinatory expressions in λ-calculus is a **simulation**. This means that for every step $c \to d$, there is a corresponding reduction $M \to_\beta^* N$ where $d \mapsto N$, possibly in multiple steps. In a picture:

$$\begin{array}{ccc} c & \mapsto & M \\ \big\downarrow & & \vdots\,\beta \\ d & \mapsto & N \end{array}$$

We can similarly observe that the interpretation $E(-)$ of λ-terms as combinatory expressions is a simulation: if $M \to_\beta N$ then $E(M) \to^* E(N)$.

$$\begin{array}{ccc} M & \xrightarrow{E} & E(M) \\ \beta\,\big\downarrow & & \vdots \\ N & \xrightarrow{E} & E(N) \end{array}$$

This is a necessary part of the interpretation being correct. But it is not sufficient: if we map every λ-term to I, then since $\to^*$ is reflexive, this would still be a simulation. The second part is that $\mapsto$ and $E(-)$ must be **inverse** to one another in some sense. In this case, we want the following: if we take a λ-term M to $E(M)$, and then back with $E(M) \mapsto N$, we want to get back essentially the same term, $M =_{\alpha\beta\eta} N$.

$$M \xrightarrow{E} E(M) \;\mapsto\; N \qquad M \overset{=_{\alpha\beta\eta}}{\frown} N$$

A correct interpretation needs to be a simulation and needs to have such an inverse.