Assignment #1

Due date: Thu. 1/16/2020, 10:00am

Assumptions

The questions below assume the system described in Handout #1 from class. Specifically, they focus on three kinds of evaluation steps described there: `let` reduction, lambda reduction, and `case` reduction. The relevant rules are repeated here.

$$\texttt{let } v \texttt{ = } e_1 \texttt{ in } e_2 \Longrightarrow [e_1/v]e_2$$

$$\texttt{(\textbackslash} v \texttt{ -> } e\texttt{) } e_2 \Longrightarrow [e_2/v]e$$
$$\texttt{(\textbackslash} v \texttt{ -> } e\texttt{) \$ } e_2 \Longrightarrow [e_2/v]e$$

$$\texttt{case Rock of \{Rock -> } e_1\texttt{; Paper -> } e_2\texttt{; Scissors -> } e_3\texttt{\}} \Longrightarrow e_1$$
$$\texttt{case Paper of \{Rock -> } e_1\texttt{; Paper -> } e_2\texttt{; Scissors -> } e_3\texttt{\}} \Longrightarrow e_2$$
$$\texttt{case Scissors of \{Rock -> } e_1\texttt{; Paper -> } e_2\texttt{; Scissors -> } e_3\texttt{\}} \Longrightarrow e_3$$

$$\texttt{case Draw of \{Draw -> } e_1\texttt{; Win } v \texttt{ -> } e_2\texttt{\}} \Longrightarrow e_1$$
$$\texttt{case (Win } e\texttt{) of \{Draw -> } e_1\texttt{; Win } v \texttt{ -> } e_2\texttt{\}} \Longrightarrow [e/v]e_2$$

Let's also suppose that the names `n`, `f`, `g` and `whatItBeats` have been defined via the following code in a Haskell file that we have loaded:

```
n = 1
f = \s -> case s of {Rock -> 334; Paper -> 138; Scissors -> 99}
g = \z -> z + 4
whatItBeats = \s -> case s of {Rock -> Scissors; Paper -> Rock; Scissors -> Paper}
```

1 Practice with evaluation

Your task here is to show how the evaluation[1] of the following expressions proceeds, one "step" at a time, i.e. showing all intermediate expressions. (See more instructions and examples below.)

1. `let x = 4 + 5 in (3 * x)`
2. `(\x -> 3 * x) (4 + 5)`
3. `((\x -> (\y -> x + (3 * y))) 4) 1`

[1] By "evaluation" here I mean "complete evaluation": take all the evaluation steps that are possible, i.e. don't just write `3 + (4 * 5)` $\Longrightarrow$ `3 + 20` and omit the further step to `23`.

4. `let x = 4 in (let y = 1 in (x + (3 * y)))`
5. `let x = 4 in (let y = 1 + x in (x + (3 * y)))`
6. `((\x -> (\y -> x + (3 * x))) 4) 1`
7. `((\x -> (\y -> y + (3 * y))) 4) 1`
8. `(\y -> y + ((\y -> 3*y) 4)) 5`
9. `(\y -> ((\y -> 3*y) 4) + y) 5`
10. `(\x -> x * (let x = 3*2 in (x + 7)) + x) 4`
11. `g ((let x = 4 in (\y -> x + y)) 2)`
12. `let x = 5 in (\z -> x * z)`
13. `(\x -> (\z -> x * z)) 5`
14. `f ((\fn -> fn Rock) (\x -> whatItBeats x))`
15. `((\f -> (\x -> f (f x))) whatItBeats) Paper`
16. `whatItBeats (case Paper of {Rock -> Paper; Paper -> Rock; Scissors -> Scissors})`
17. `(case (Win Rock) of {Draw -> whatItBeats; Win z -> (\s -> Scissors)}) Paper`
18. `case (Win (whatItBeats Rock)) of {Draw -> n; Win x -> (n + f x)}`
19. `let y = 2 in (case (Win (whatItBeats Rock)) of {Draw -> n; Win y -> (n + f y)} + y)`

For the purposes of this exercise, we'll take one step to be either:

- a `let` reduction step,
- a lambda reduction step,
- a `case` reduction step,
- a substitution using a definition from our loaded file (much like a `let` reduction step), or
- a single arithmetic addition or multiplication operation.

Label each intermediate expression in your answer to indicate which kind of step is being taken. Here are two examples:

	`(\x -> (3 + x) * n) 2`	
$\Longrightarrow$	`(3 + 2) * n`	lambda reduction
$\Longrightarrow$	`5 * n`	arithmetic
$\Longrightarrow$	`5 * 1`	substitution from file
$\Longrightarrow$	`5`	arithmetic

	`let z = Paper in (f (whatItBeats z))`	
$\Longrightarrow$	`f (whatItBeats Paper)`	let reduction
$\Longrightarrow$	`f ((\s -> case s of {Rock -> Scissors; Paper -> Rock; Scissors -> Paper}) Paper)`	substitution from file
$\Longrightarrow$	`f (case Paper of {Rock -> Scissors; Paper -> Rock; Scissors -> Paper})`	lambda reduction
$\Longrightarrow$	`f Rock`	case reduction
$\Longrightarrow$	`(\s -> case s of {Rock -> 334; Paper -> 138; Scissors -> 99}) Rock`	substitution from file
$\Longrightarrow$	`case Rock of {Rock -> 334; Paper -> 138; Scissors -> 99}`	lambda reduction
$\Longrightarrow$	`334`	case reduction

A couple of things to note:

- You can check (most of!) the final results using ghci if you wish, but you will be graded on getting the intermediate steps correct as well.

- In many cases there are multiple routes to the final result, depending on which part of the expression you choose to simplify first. You'll get the same result no matter which route you take, but some routes involve more work than others.

2 Constructing expressions yourself

Of course "doing programming" does not involve evaluating expressions, but rather involves *constructing* expressions that will have certain desired effects when they are evaluated. The aim of this exercise is to give you a taste of how that relates to the exercises above. It might sound much more complicated at first than it really is.

Your task here is to construct an expression e, whose value depends on a shape represented by the variable `s` and a number represented by the variable `x`, such that:

- if the shape `s` is `Rock`, then e evaluates to the square of the number `x`; and
- if the shape `s` is `Paper`, then e evaluates to the cube of the number `x`; and
- if the shape `s` is `Scissors`, then e evaluates to the number `x`.

To confirm[2] that your chosen expression e does the right thing, show that:

- $[\texttt{Rock}/\texttt{s}][4/\texttt{x}]e \Longrightarrow \texttt{16}$
- $[\texttt{Paper}/\texttt{s}][4/\texttt{x}]e \Longrightarrow \texttt{64}$
- $[\texttt{Scissors}/\texttt{s}][4/\texttt{x}]e \Longrightarrow \texttt{4}$

Note that e will need to be an *open* expression.

[2] Of course, this isn't actually a *proof*, ...