Ling 185a: Assignment 1

Dylan Bumford, Winter 2024

Assumptions

The questions below assume the evaluation system described in class. Specifically, they focus on the evaluation of `let`, lambda, and `case` expressions, the rules for which are repeated below.

- `let v = exp in body` $\Longrightarrow$ [`v` $\mapsto$ `exp`]{`body`} — `let` reduction
- `(\v -> body) exp` $\Longrightarrow$ [`v` $\mapsto$ `exp`]{`body`} — lambda reduction
- `case Con of {...; Con -> body; ...}` $\Longrightarrow$ `body` — `case` reduction
  `case Con exp of {...; Con v -> body; ...}` $\Longrightarrow$ [`v` $\mapsto$ `exp`]{`body`} — `case` reduction

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

```
n = 1
f = (\s -> case s of {Rock -> 112; Paper -> 71; Scissors -> 304})
losesTo = (\s -> case s of {Rock -> Scissors; Paper -> Rock; Scissors -> Paper})
```

Practice with evaluation

Your task here is to evaluate the following expressions one step at a time, showing all intermediate expressions until the result cannot be further evaluated. (See more instructions and examples below.)

a. `let x = 4 + 5 in 3 * x`

b. `(\x -> 3 * x) (4 + 5)`

c. `((\x -> (\y -> x + (3 * y))) 4) 1`

d. `let x = 4 in (let y = 1 + x in (x + (3 * y)))`

e. `(\y -> y + ((\y -> 3 * y) 4)) 5`

f. `(\y -> ((\y -> 3 * y) 4) + y) 5`

g. `(\x -> x * (let x = 3 * 2 in (x + 7)) + x) 4`

h. `let k = (\x -> (let y = 3 in x + y)) in k 4`

i. `let k = (let y = 3 in \x -> x + y) in k 4`

j. `f ((\k -> k Rock) (\x -> losesTo x))`

k. `((\f -> (\x -> f (f x))) losesTo) Paper`

l. `losesTo (case Paper of {Rock -> Paper; Paper -> Rock; Scissors -> Scissors})`

m. `case MyMove (losesTo Paper) of {YourMove v -> n; MyMove x -> (n + f x)}`

n. `(case MyMove Rock of {YourMove v -> losesTo; MyMove z -> (\s -> Scissors)}) Paper`

o. `let y = 2 in (case MyMove (losesTo Rock) of {YourMove v -> n; MyMove y -> (n + f y)} + y)`

For the purposes of this exercise, a step is one of the following:

- a `let` reduction
- a lambda reduction
- a `case` reduction
- a single arithmetic operation
- a replacement of an identifier by its definition

Label each intermediate expression in your answer to indicate which of these evaluation rules got you there from the previous expression. Here are two examples:

```
                (\x -> (3 + x) * n) 2
lambda       ⟹ (3 + 2) * n
arithmetic   ⟹ 5 * n
def. of n    ⟹ 5 * 1
arithmetic   ⟹ 5
```

```
                 let z = Paper in (f (losesTo z))
let           ⟹ f (losesTo Paper)
def. of losesTo ⟹ f ((\s -> case s of {Rock -> Scissors; Paper -> Rock; Scissors -> Paper}) Paper)
lambda        ⟹ f (case Paper of {Rock -> Scissors; Paper -> Rock; Scissors -> Paper})
case          ⟹ f Rock
def. of f     ⟹ (\s -> case s of {Rock -> 112; Paper -> 71; Scissors -> 304}) Rock
lambda        ⟹ case Rock of {Rock -> 112; Paper -> 71; Scissors -> 304}
case          ⟹ 112
```

A few things to note:

- You can check 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 value, 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.
- If you have something like `(\s -> case s of {...}) Paper`, then taking one step of evaluation leads you to `case Paper of {...}` (don't do the case reduction and the lambda reduction in a single step)
- Be very careful with parentheses. Whenever you replace an identifier with an expression `exp` in the `body` of some binding term, you need to make sure `exp` is treated like a constituent in `body`. This means that if there could be any ambiguity at all, you should put parentheses around `exp` to keep it grouped together. This is especially true when `exp` itself is a binding expression (a `let`, lambda, or `case`).