CS4400/5400 Spring 2024

Assignment 5: Untyped and typed lambda calculi

Objectives:

- Become comfortable evaluation strategies for the untyped lambda calculus.
- Understand the role of a typechecker in programming.

Resources:

- The Plait tutorial and documentation page: https://docs.racket-lang.org/plait/

Parameters:

- Due date: Wednesday Feb. 21 at 11:59PM
- Upload your solution **as a** `code.rkt` **file** on Gradescope.
- Use the following starter code
- Ask questions on Piazza or during Office Hours.
- Perform all the work for this project by yourself. You may discuss with your classmates, but you must write your own code.
- Grading: each problem is worth a certain number of points. The points given for each problem is the fraction of tests passed for that problem (i.e., if 50% of the tests are passed for a 20 point problem, 10 points are given). Note that this week's homework will have some non auto-graded portions, so your final score will not be determined solely by the autograded score.

Problem 1. Call me by your name (or value) [40 Points]

In class we've seen the λ-term Ω, which runs forever:

$$\Omega = (\lambda x.\ x\ x)(\lambda x.\ x\ x)$$

It runs forever because, once you do function application, you still end up with Ω–no progress has been made. Such programs are called *divergent*.

It's reasonable to think that, if you have Ω anywhere inside your program, then it'll diverge. In this exercise we'll show that's not necessarily the case!

The key to why is to understand what an *evaluation strategy* is. An evaluation strategy is simply *how* you evaluate a program into a value.

For example, consider the following Plait program:

```
((lambda (x)  (* x 10)) (+ 2 5))
```

where we are applying an anonymous function to the input `(+ 2 5)`. There are two ways to evaluate this. In the first way, which is the one we've been seeing in class, we can evaluate the argument `(+ 2 5)` *before* substitution:

```
; Evaluting (+ 2 5) before applying the lambda
((lambda (x)  (* x 10)) (+ 2 5))
--> ((lambda (x) (* x 10)) 7)    ; first evaluate argument
--> (* x 10)[x |-> 7]            ; then substitute argument into the body
--> (* 7 10)                     ; then evaluate the body
--> 70
```

An alternative way to evaluate this program is to substitute the argument *before* evaluating it:

```
((lambda (x)  (* x 10)) (+ 2 5))
--> (* x 10)[x |-> (+ 2 5)]     ; substitute the argument before evaluating
--> (* (+ 2 5) 10)              ; then evaluate the body
--> (* 7 10)
--> 70
```

We call the first evaluation strategy *call-by-value*, and the second *call-by-name*. This is one of the first design choices programming language designers have to make.

How do these strategies help us in taming the divergent nature of Ω? First let's set up our abstract syntax for the lambda calculus:

```
(define-type lcalc
  (var [s : Symbol])
  (lam [s : Symbol] [b : lcalc])
  (app [fn : lcalc] [b : lcalc]))

; We'll be using Omega a lot, so let's define it here.
(define Omega (app (lam 'x (app (var 'x) (var 'x))) (lam 'x (app (var 'x) (var 'x)))))
```

And let's see our call-by-name and call-by-value strategy in action, on the λ-term $(\lambda z.\ \lambda w.\ w)\Omega$:

```
; Call-by-value
(app (lam 'z (lam 'w (var 'w))) Omega)
--> (app (lam 'z (lam 'w (var 'w))) Omega) ; Omega gets reduced...to Omega...
--> (app (lam 'z (lam 'w (var 'w))) Omega); and again...
--> (app (lam 'z (lam 'w (var 'w))) Omega) ; and again...
...
```

```
; Call-by-name
(app (lam 'z (lam 'w (var 'w))) Omega)
--> (lam 'w (var 'w))[z |-> Omega] ; the function is applied before input is reduced
--> (lam 'w (var 'w)) ; and Omega magically disappears!
```

Moral of the story is that the call-by-value strategy evaluates inputs to a function before applying it, and the call-by-name strategy evaluates function application before the input.

Your job is now to implement the two evaluation strategies.

Problem 1a. Write a function `call-by-value` of type `(lcalc -> lcalc)` that takes in a λ-calculus expression and evaluates it via the call-by-value evaluation strategy.

Problem 1b. Write a function `call-by-name` of type `(lcalc -> lcalc)` that takes in a λ-calculus expression and evaluates it via the call-by-name evaluation strategy.

Problem 1c. Write a λ-term `example` of type `lcalc` such that the test `(test (equal? (call-by-name example) (call-by-value example)) #f)` passes. If such an example does not exist, briefly explain why in a comment.

Problem 2: Derivations practice [10 Points]

Consider the following tiny language for a calculator language with conditionals, let-bindings, true, and false:

```
τ ::= Number | Bool       ; set of possible types
<e> ::= (if <e> <e> <e>)
       | (let (x : τ <e>) <e>)
       | num
       | true
       | false
```

Let's build a tiny type-system for this language. Let's define our context Γ to be a map from variables to their types. We will use the following rules:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash \texttt{x} : \tau}\text{(T-var)}$$

$$\frac{}{\Gamma \vdash \texttt{num} : \texttt{Number}}\text{(T-Num)} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}\text{(T-true)} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}\text{(T-false)}$$

$$\frac{\Gamma \vdash \texttt{e1} : \texttt{Bool} \qquad \Gamma \vdash \texttt{e2} : \tau \qquad \Gamma \vdash \texttt{e3} : \tau}{\Gamma \vdash \texttt{(if e1 e2 e3)} : \tau}\text{(T-If)}$$

$$\frac{\Gamma \vdash \texttt{e}_1 : \tau 1 \qquad \Gamma \cup \{x \mapsto \tau\} \vdash \texttt{e}_2 : \tau 2}{\Gamma \vdash \texttt{(let (x : τ1 e1) e2)} : \tau 2}\text{(T-Let)}$$

Give typing derivation trees for the following terms:

`(if true 10 20)`

`(let (x : Number 2) (if true x x))`

`(let (x : Bool true) (let (y : Number 1) (if x y y)))`

Problem 3: Building a type-checker [40 Points]

Recall the simply-typed λ-calculus (STLC) type-checker we made in class. Let's consider an extension of the STLC *with conditionals*:

```
(define-type LType
  [NumT]
  [FunT (arg : LType) (body : LType)])

(define-type LExp
  [varE (s : Symbol)]
  [numE (n : Number)]
  [ifE (g : LExp) (thn : LExp) (els : LExp)]
  [lamE (arg : Symbol) (typ : LType) (body : LExp)]
  [appE (e : LExp) (arg : LExp)])
```

Problem 3a: Make a function `interp` of type `LExp -> LExp` for the simply-typed lambda calculus with conditionals. The semantics of `ifE` should be the usual one we've seen in class for conditionals with numbers: if the guard is `0`, then evaluate the `thn` branch; if the guard is non-zero, evaluate the `els` branch; otherwise, raise a `'runtime` error. You should base your interpreter on the one we went over in class. Examples:

```
> (interp (appE (lamE (quote x) (NumT) (varE (quote x))) (numE 10)))
- LExp
(numE 10)

> (interp (ifE (numE 0) (lamE 'x (NumT) (varE 'x)) (lamE 'x (NumT) (numE 10))))
- LExp
(lamE 'x (NumT) (varE 'x))
```

Problem 3b: Give 2 programs that "go wrong", i.e. cause your interpreter in 3a to fail with a runtime error. You should add these errors to your `code.rkt` file as `text/exn` functions.

Problem 3c: Give typing rule(s), written as inference rule(s), for the simply-typed λ-calculus with conditionals that would eliminate the typing errors you found in 3a.

Problem 3d: Write a typechecker `type-of` of type `LExp -> LType`; your `type-of` function should raise

a `type-error` if there is no valid type. Your typechecker should ensure that any well-typed program cannot cause your interpreter in 3a to raise a contract or runtime error. You will need a helper function.