Background

In this assignment you will implement versions of LFP with
(i) call-by-value evaluation and dynamic scoping,
(ii) call-by-name evaluation and lexical scoping, and
(iii) call-by-name evaluation and dynamic scoping.

The version of LFP we’ll be working with has two new commands:
1. The return command works pretty much as it does in C.

\[
\text{Return: } \rho \vdash (e, s) \downarrow (v, s')
\]

\[
\rho \vdash \text{return } e, s \downarrow (v, s')
\]

2. The print command does what you expect — sort of.

\[
\text{Print: } \{ \rho \vdash (e_1, s_1) \downarrow (v_1, s_1+1) \}_{i=1, \ldots, n} \left( \text{the values of } e_1, \ldots, e_n \right)
\]

\[
\rho \vdash \text{print } (e_1, e_2, \ldots, e_n), s_1 \downarrow \text{skip}, s_{n+1} (\text{the values of } e_1, \ldots, e_n \text{ are printed})
\]

The print command is implemented on the cheap (i.e., via Haskell’s trace function) and so print’s can show up in the output in very odd places and in very odd orders because of Haskell’s laziness. (Try evaluating et0a and et0b to see what I mean.)

The big-step operational semantics rules for (i) Call-by-value, lexical-scoping, (ii) Call-by-value, dynamic-scoping, (iii) Call-by-name, lexical-scoping, and (iv) Call-by-name, dynamic-scoping are given below. I’ve put closures in boxes. Also, the hat in \( \hat{\rho} \) is just a distinctive decoration.

Call-by-value, lexical-scoping

\[
\rho \vdash e_1, s \downarrow v \langle (\lambda x.e_1) \hat{\rho}_1, s' \rangle
\]

\[
\rho \vdash e_2, s' \downarrow v \langle v, s'' \rangle
\]

App: \[
\hat{\rho}_1 [x \mapsto v_2] \vdash (\hat{\epsilon}_1, s''') \downarrow v \langle v, s''' \rangle
\]

\[
\rho \vdash (e_1 e_2), s \downarrow v \langle v, s''' \rangle
\]

Var: \[
\rho \vdash (x, s) \downarrow v \langle v = \text{lookup}(\rho, x) \rangle
\]

Fun: \[
\rho \vdash (\lambda x.e, s) \downarrow v \langle (\lambda x.e) \rho, s \rangle
\]

Call-by-value, dynamic-scoping

\[
\rho \vdash e_1, s \downarrow v \langle (\lambda x.e_1'), s' \rangle
\]

\[
\rho \vdash e_2, s' \downarrow v \langle v_2, s''' \rangle
\]

App: \[
[\rho [x \mapsto v_2] \vdash e_1', s'''] \downarrow v \langle v, s''' \rangle
\]

\[
\rho \vdash (e_1 e_2), s \downarrow v \langle v, s''' \rangle
\]

Var: \[
\rho \vdash (x, s) \downarrow v \langle v = \text{lookup}(\rho, x) \rangle
\]

Fun: \[
\rho \vdash (\lambda x.e, s) \downarrow v \langle (\lambda x.e), s \rangle
\]

Call-by-name, lexical-scoping

\[
\rho \vdash e_1, s \downarrow N \langle (\lambda x.e_1) \hat{\rho_1}, s' \rangle
\]

\[
\hat{\rho}_1 [x \mapsto e_2\rho] \vdash (\hat{\epsilon}_1, s') \downarrow N \langle v, s'' \rangle
\]

\[
\rho \vdash (e_1 e_2), s \downarrow N \langle v, s'' \rangle
\]

Var: \[
\rho \vdash (x, s) \downarrow N \langle v, s' \rangle \left( e' = \text{lookup}(\rho, x) \right)
\]

Fun: \[
\rho \vdash (\lambda x.e, s) \downarrow N \langle (\lambda x.e) \rho, s \rangle
\]

Call-by-name, dynamic-scoping

\[
\rho \vdash e_1, s \downarrow N \langle \lambda x.e_1', s' \rangle
\]

\[
\rho [x \mapsto e_2] \vdash e_1', s' \downarrow N \langle v, s'' \rangle
\]

\[
\rho \vdash (e_1 e_2), s \downarrow N \langle v, s'' \rangle
\]

Var: \[
\rho \vdash (x, s) \downarrow N \langle v, s' \rangle \left( e = \text{lookup}(\rho, x) \right)
\]

Fun: \[
\rho \vdash (\lambda x.e, s) \downarrow N \langle (\lambda x.e), s \rangle
\]
The file `hw10.hs` has the call-by-value/lexical-scoping version of LFP. You will also need `LFP2.hs`, `Parser3.hs`, and `State.hs`.

**Problems**

> **Problem 1:** (20 points).

> **Problem 2:** (20 points).

> **Problem 3:** (20 points).
Change the version of LFP of `hw11.hs` to call-by-name/dynamic-scoping. Test your program on `et1`, `et2`, `et3`, `et4`, and `et5`.

> **Problem 4:** (40 points).
For each of the following LFP programs, figure out by hand what the program prints or returns. You can use your code to check your work, but working out these by hand will give you some practice for the next quiz.

(a) Consider:

```haskell
let x = 1
in let f = (λz.(x + z))
in let x = 100 in (f 10)
```

(i) (4 points) What is returned under call-by-value/lexical scoping?
(ii) (4 points) What is returned under call-by-value/dynamic scoping?

(b) Consider:

```haskell
let x = 100
in let f = (λu.(u * x))
in let g = (λx.(f 2))
in print ((f 10), (g 6))
```

(i) (4 points) What is printed under call-by-value/lexical scoping?
(ii) (4 points) What is printed under call-by-value/dynamic scoping?

(c) Assume the location `X1` (or `ℓ1` if you prefer) starts out with contents 0. Consider:

```haskell
let f = λy. { X1 :=! X1 + 10; return y }
in let y = (f 40)
in { X2 := y + 2 * y; return (val (X2) + val (X1)) }
```

(i) (4 points) What is returned under call-by-value/lexical scoping?
(ii) (4 points) What is returned under call-by-name/lexical scoping?

(d) Assume `X1` starts out with contents 0. Consider:

```haskell
let f = λy. { X1 :=! X1 + 100; return y }
in let g = λz.10
in let w = (g (f 999)) in (w + val (X1))
```

(i) (4 points) What is returned under call-by-value/lexical scoping?
(ii) (4 points) What is returned under call-by-name/lexical scoping?

(e) Assume `X1` starts out with contents 0. Consider:

```haskell
let tick = 1
in let tock = λu. { X1 := val(X1) + tick; return val (X1) }
in let tick = 100 in (tock (tock (tock 0)))
```

(i) (2 points) What is returned under call-by-value/lexical scoping?
(ii) (2 points) What is returned under call-by-value/dynamic scoping?
(iii) (2 points) What is returned under call-by-name/lexical scoping?
(iv) (2 points) What is returned under call-by-name/dynamic scoping?

**Administrivia**

This assignment is a solo effort: NO TEAMS!

> Turn in problem 4 in the CIS 352 submissions box. If you trade ideas with another student, document this in your cover sheet.
> For problems 1, 2, and 3, turn them in via Blackboard. Include:
  (i) the source files,
  (ii) the transcripts of test runs, and
  (iii) your cover sheet.