writeup.md

part 1: The substitution model

I ran into dead ends a few times as a result of my misunderstanding of the spec. For example I thought `f f 3` would implicitly be the same as `f (f 3)`. I.e application would work its way from right to left. I initially thought this because `let f = fun x -> x in f f 3` was given as an example in the project spec. However, this is a special case since f is the identity function. `f f 3` Should return an error for any function f expecting an int. To do this correctly we actually want to evaluate `f (f 3)`. TLDR parenthesis are important.

part 2: The dynamic environment model

I did not have as much trouble with this section of the project. Although for a while I was using the wrong environment when evaluating App. I was subsituting into the environment associated with the first element in the App, rather than the environment which was passed to eval_d with the App. This caused some annoying bugs. It is also important to note that the dynamic model will return different results from the substitution for certain expressions. This happens when when variables are defined in multiple places. In a dynamic environment the variables used within a function are decided when the function is applied. In the substitution model the function uses the variables which are defined when the functions are defined. For example, a function which will return different results with the two models:

```
Subst:
<== let x = 1 in let f = fun y -> x + y in let x = 2 in f 0;;
==> 1

Dynamic:
<== let x = 1 in let f = fun y -> x + y in let x = 2 in f 0;;
==> 2
```

and a function which will return the same result in both models.

```
Subst:
<== let x = 1 in let x = 2 in let f = fun y -> y * x in f 5;;
==> 5
Dynamic:
<== let x = 1 in let x = 2 in let f = fun y -> y * x in f 5;;
==> 10
```

part 3: Lexical environment model

The lexical environment model evaluates expressions within the context of the environment they are defined in. In this way it is like the substitution model but more flexible, because if we extened the interpreter to have persistent state, or in other words keep track of the environment then we could do the following.

```
<== let f = fun x -> x + x;;
<== f 5;;
==> 10
```

If I had more time I would have liked to implement this extension.

The implementation of this is fairly similar to the dynamic environment. One of the obvious differences is the large recursive function in eval_l called heval_l. This is necessary because I decided that all of the eval functions should return an expr, and it was necessary to make extensive use of values in eval_l. In eval_l I call heval_l on the input and match the output on heval_l to return an expr.

The core of heval_l is a match statement on expr and then recursively calling heval_l on subcomponents. Heval_l eventually returns a Env.Val for except for functions where we return a Env.Closure of the function and current environment. This is

because we are implementing a lexical scope and want to keep track of the state of the program at the moment each function is defined.

evaluating let rec statements is also tricky. If we have `Letrc(id, recfun, e2)`, then we want to create a new recusive function by evaluating the old one in a environment with id set to Unassigned. I used eval_l here because I just needed an expr. We complete the implementation of let rec by evaluating e2, which in simple cases could just be an application, in a environment where id maps to the new recursive function. What this is effectively doing is taking a letrec and turning it into a function application with a modified function

App is the most important part of this implmentation. We want to evaluate and match the left side of `App(e1, e2)` in the passed env. We only match on Closure because it is the only way a function will be represented as a value. If it is not a Closure then we raise an exception because we can only apply functions. Since we have the closure returned from using heval_l on a function we now have both the function and the environment in which it was defined. So now we can evaluate this function by evaluating the body in the environment with the extension of setting the input variable to the applied expr.

The other possible forms of expression are fairly straightforward. First evaluate any sub expressions and then apply the relevent rules of the constructs to the evaluated expresions.