CS 51 Final Project
MiniML --Write_UP
Spring 2019
Tito Yak

part 1: The substitution model

My intial difficulty lied with my misunderstanding of specs. For example I assumed that `f f 3` would implicitly be the same as `f (f 3)`. I.e tghe application of the function would would evaluate to its output from right to left. I made this wrong conclusion beause `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 other function f. So in a nutshell, the parenthesis eliminate the ambiguity. We should evaluate `f (f 3)`.

part 2: The dynamic environment model

This was the most challenging part for me. I almost gave up implementing letrec. I also struggled with type. To solve this, I used the same design I had for substitution model. I created a helper function(heval_d) which saved me from stress. I noted that 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:

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

Dynamic environment model:
<== 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.

```
Substitution model:
<== let x = 1 in let x = 2 in let f = fun y -> y * x in f 5;;
==> 5
Dynamic environment model:
<== 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
```

My implementation here is quite similar to that of dynamic environment. One of the few differences is the large recursive function in eval_l called heval_l. In eval_l I call heval_l on the input and match the output on heval_l to return an expr.

The prime spine 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 every type of exp 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 quite tricky. With `Letrc(id, recfun, xp2)`, we create a new recusive function by evaluating the old one in a environment with id set to Unassigned. We complete the implementation of let rec by evaluating xp2, 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

For App, We want to evaluate and match the left side of $\mathrm{App}(\mathrm{xp1}, \mathrm{xp2})$ 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.

For other possible forms of expression, first we would evaluate any sub-expressions and then apply the relevent rules of the constructs to the evaluated expresions.