My Final CS51 Project Extension

Extension Implemented: Lexical Environment Semantics

For my project extension, I decided to implement a lexical environment semantics evaluator on top of the dynamic environment semantics evaluator I had already coded for the project.

Initially, I had not been extremely confident in my understanding of lexical vs dynamic semantics, however after watching Schieber's impromptu lecture and studying the rules that were given in the textbook, I was able to realize that lexical semantics is simply "executing the code as you would read." I figured out that in lexical semantics, when lines of code were reached–such as a function definition–they would be defined then and there in the environment that the function was in when the function was ***defined***. This differs from dynamic evaluation where the function is defined using the environment it is in when it is ***applied***. Because this is the main difference between dynamic and lexical, I knew that everything else about the evaluators would be the same except the Fun and App match case for the expressions fed into the evaluators. Thus, I created a general eval_env function and moved all the repeated code from the dynamic and lexical evaluators into there. Then, in order to help that general evaluator distinguish between the dynamic Fun and App and the lexical Fun and App, I created an abstract data type called ***semantic_type*** which had two data types: dynamic and lexical. Then I pattern matched for these new data types within the Fun and App match cases to determine which function or application implementation I should use. (screenshot attached)

```
type semantic_type =
  | Dynamic
  | Lexical
```

```
| Fun (x, expr) ->
  (match semantic_type with
  | Dynamic -> Val(Fun(x, expr))
  | Lexical -> Env.close exp env)
| App (funexpr, expr) ->
  match semantic_type with
  | Dynamic ->
    (match eval_env funexpr env semantic_type with
    | Env.Val (Fun (varid, fundef)) ->
      let vQ = eval_env expr env semantic_type in
      eval_env fundef (Env.extend env varid (ref vQ)) semantic_type
    | _ -> raise (EvalError "didn't input a function"))
  | Lexical ->
    (match eval_env funexpr env semantic_type with
    | Env.Closure (Fun (x, lexpr), lexicalenvironment) ->
      let vQ = eval_env expr env semantic_type in
      eval_env lexpr (Env.extend lexicalenvironment x (ref vQ)) semantic_type
    | _ -> raise (EvalError "didn't input a function")) ;;
```

A demonstration of the difference between the dynamical and lexical environment that I implemented is clearly shown through the screenshot below.

```
[paulsong@dhcp-10-250-123-139 finalproject % make all
ocamlbuild -use-ocamlfind expr.byte
Finished, 5 targets (5 cached) in 00:00:00.
ocamlbuild -use-ocamlfind evaluation.byte
Finished, 7 targets (4 cached) in 00:00:00.
ocamlbuild -use-ocamlfind miniml.byte
Finished, 18 targets (16 cached) in 00:00:00.
ocamlbuild -use-ocamlfind miniml_tests.byte
Finished, 20 targets (18 cached) in 00:00:00.
[paulsong@dhcp-10-250-123-139 finalproject % ./miniml.byte
<== let x = 1 in let f = fun y -> x + y in let x = 2 in f 3 ;;
==> 5
<== Goodbye.
[paulsong@dhcp-10-250-123-139 finalproject % make all
ocamlbuild -use-ocamlfind expr.byte
Finished, 5 targets (5 cached) in 00:00:00.
ocamlbuild -use-ocamlfind evaluation.byte
Finished, 7 targets (4 cached) in 00:00:00.
ocamlbuild -use-ocamlfind miniml.byte
Finished, 18 targets (16 cached) in 00:00:00.
ocamlbuild -use-ocamlfind miniml_tests.byte
Finished, 20 targets (18 cached) in 00:00:00.
[paulsong@dhcp-10-250-123-139 finalproject % ./miniml.byte
<== let x = 1 in let f = fun y -> x + y in let x = 2 in f 3 ;;
==> 4
<== Goodbye.
paulsong@dhcp-10-250-123-139 finalproject %
```

The first execution of the code is in dynamic semantics and the second evaluation is in lexical.

In the first case, the function uses the environment which maps x to the number 2 to evaluate f whereas in the second case, the function uses the environment where x mapped to 1 to evaluate f to be "fun y -> 1 + y". The reaons why is because in dynamic, the x is changed before the function is applied and thus looking up the value of x. In lexical, the x is not changed after it is set to 1, and then the function is mapped to "fun y -> 1 + y" right after. Even though x changes after the function definition, it does not affect the function f because the function f was closed in a closure, kind of like a snapshot in time of the environment.