CS51 Final Project Writeup

Ben Kaplan

1 Lexical Evaluation

For my final project extension I decided to implement an additional evaluate function for lexical scoping. I felt that it would be a good extension to my project as it would show the distinction between OCaml (dynamically scoped) and other languages that are lexically scoped. This is highlighted by some of my test cases that return different values when evaluated with $eval_d$ as opposed to $eval_l$.

Example 1:

```
let x = 3 in
let rec f = fun y ->
  if y = 0 then 1
  else x * f (y - 1) in
let x = 2 in f 4;;
```

The dynamic evaluation of this code will first associate with the variable x the value 3 (x, 3). Then it will override the value of x, and replace it with (x, 2). When the function evaluates f 4 it will then search in the environment for value of x which has been overridden as (x, 2) and then return 16. In contrast, lexical scoping with environments or substitution will maintain the initial value of x = 3 and create a closed environment in which whenever the function f is evaluated, it will look up the value of x and find that (x, 3) in this environment in which f is evaluated. Therefore f 4 will return $3 * f(3) = 3 * 3 * f(2) = 81$.

In order to implement lexical scoping I first copied over my dynamic evaluation function into a new function. I realized that my unary and binary operator evaluation remained the same so I abstracted that out of the code so that I did not copy and paste the same code from $eval_s$ to $eval_d$ to $eval_l$. I also had to now use the closures of type Value to make smaller subset environments to ensure that a function similar to the f above, would evaluate in an environment where x = 3 and would not have the x value be overridden as 2. I therefore had to rewrite the match cases for fun(x, e), letrec (x, e, e1), and app(e, e1). In rewriting these match cases I was able to take into account the creation of closures to create these small subset environments something that we did not do in $eval_d$. As a whole it was interesting to see the different ways of evaluating code even within the environment method to allow for different results

depending on if the code was lexically or dynamically evaluated. The dynamic evaluation of this code will first associate with the variable x the value 3 (x, 3). Then it will override the value of x, and replace it with (x, 2). When the function evaluates f 4 it will then search in the environment for value of x which has been overridden as (x, 2) and then return 16. In contrast, lexical scoping with environments or substitution will maintain the initial value of x = 3 and create a closed environment in which whenever the function f is evaluated, it will look up the value of x and find that (x, 3) in this environment in which f is evaluated. Therefore f 4 will return $3 * f(3) = 3 * 3 * f(2) = 81$.

The distinction in how the different evaluation methods work is clearly seen when running the test cases. The substitution evaluator, returns the same values as the lexical evaluator as it creates new variable names when it finds a repeat free variable and therefore prevents the code from overwriting itself. I therefore ran all of my $eval_s$ test cases on $eval_l$ and my code worked properly in returning the same values. When I ran the same tests on the dynamic evaluator some of the tests that did not contain variable values being overridden did pass, however, the ones that did contain instances including Example 1 failed to evaluate to the same value, as the value that the code returns is different when dynamically vs lexically evaluated. Another example of a simpler non-recursive function that returns different values when dynamically evaluated as opposed to lexically evaluated is Example 2.

Example 2:

```
let x = 7 in
let f = fun y -> x * y in
let x = 5 in
f 8 ;;
```

In example 2, the x value has the potential to be overridden. In dynamic evaluation the x value will be overridden and the evaluator will return 40. In a lexically scoped evaluation, the second x will shadow the first, but not override it within the function f. Therefore $eval_l$ will will return 56.

2 Int Binop Functions

I initially decided to try an implement additional integer binary operators such as Division, and GreaterThan for my final project extension. I read through the various miniml files and realized that division is applied the same way the times operator is applied. I therefore scanned these files carefully and added division in every instance that I saw multiplication. This included adding the word Divide as well as the symbol '/'. I then added Divide to the type Binop in expr.mll and expr.ml. I also had to add the match case of divide in the various functions that I had written. As soon as I understood how to do this I decided to also add the GreaterThan sign (>) to my implementation of miniml. It was then very simple to add the GreaterThan sign as I had figured out every place

where the name or sign needed to be added. As I realized that it would be very quick to keep adding integer operators, I decided to implement lexical evaluation as it would be a more substantial extension. Nonetheless, I did leave the Divide and GreaterThan Binops and symbols as part of my implementation.