Final Project Writeup

Computer Science 51 2021

May 5, 2021

1 Introduction

I extended MiniML by adding a lexically scoped evaluator, floats, division, and a greater than symbol.

2 Lexical

To implement the lexically scoped evaluator, I first copied my *eval_d* function and adjusted it according to the rules for lexical semantics in the textbook. I realized that there were many similarities between *eval_d* and *eval_l*, so I decided to abstract away the similarities to avoid unnecessary code reuse and to preserve the edict of irredundancy.

My original strategy for this was to adjust *eval_l* so that it only evaluated itself for the cases in which it differed from *eval_l*, and otherwise simply called *eval_d*. However, I realized that this might not preserve the edict of intention.

I therefore decided to create a functor which, after being passed in a specific type of evaluation, would evaluate it accordingly. Therefore, the specific implementation of *eval_d* and *eval_l* are behind the abstraction barrier, and the edict of intention is preserved: it is clear to users of my code that *eval_d* and *eval_l* are different. However, the functor abstracts away their similarities to preserve the edict of irredundancy.

To demonstrate this extension, simply call *./miniml.byte* and run the repl. My repl outputs the substitution, dynamic, and lexical evaluations, so it is easy

to see how *eval_l* and *eval_d* differ by inputting an expression that will have a different output in each evaluator.

2.1 Differences Between Lexical and Dynamic Evaluations

There are various conditions in which the evaluator returns a different result under the different semantics. I walk through and demonstrate a few examples of this below.

A. In this example, we first set x equal to 2 in a function f where we take a y and output an $x * y$, and then set x equal to 1 before calling the function on 21. In the lexical semantics, the x in the definition of f refers to outer x, which is of course 2. However, in the dynamic semantics, the x in the definition of f refers to the most recent x, and is therefore 1. Therefore, the different cases return different results.
   As the substitution semantics are lexically scoped, they return the same result as the lexical semantics.

```
<== let x = 2 in let f =  fun y -> x * y in let x = 1 in f 21 ;;
--> Let (x, Num (2), Let (f, Fun (y, Binop (Times, Var (x), Var (y))), Let (x, Num (1), App (Var (f), Num (21)))))
s => 42
d => 21
l => 42
```

Figure 1: Semantics Example A

B. In this example, we first set x equal to 10 in a function f where we take a y and output another function, which takes a z and outputs a $z * (x + y)$, then calling f on 11 and 2.

   When we apply f to 11, we return a function from z to $z * (x + y)$. However, we return this out of the context in which y is defined. Therefore, the dynamic semantics will return an *unbound variable* error, while the lexical and substitution semantics will handle it just fine.

```
<== let x = 10 in let f = fun y -> fun z -> z * (x + y) in f 11 2;;
--> Let (x, Num (10), Let (f, Fun (y, Fun (z, Binop (Times, Var (z), Binop (Plus, Var (x), Var (y))))), App (App (Var (f), Num (11)), Num (2))))
s => 42
l => 42
```

Figure 2: Semantics Example B1

Here I have set the evaluator to only use dynamic semantics, returning an *unbound variable* error.

```
<== let x = 10 in let f = fun y -> fun z -> z * (x + y) in f 11 2;;
--> Let (x, Num (10), Let (f, Fun (y, Fun (z, Binop (Times, Var (z), Binop (Plus, Var (x), Var (y))))), App (App (Var (f), Num (11)), Num (2))))
xx> evaluation error: variable unbound
```

Figure 3: Semantics Example B2

C. We can correct or repair this error by ensuring that y has a binding before the function f is called. Here dynamic semantics still return something different than lexical and substitution semantics, but is able to handle this 2-argument function. Dynamic semantics will take the function as if it is being applied to 12 and 2 instead of 11 and 2, because it uses the environment in which y is equal to 12 as opposed to 11.

```
<== let x=10 in let f=fun y->fun z ->z * (x + y) in let y = 12 in f 11 2;;
--> Let (x, Num (10), Let (f, Fun (y, Fun (z, Binop (Times, Var (z), Binop (Plus, Var (x), Var (y))))), Let (y, Num (12), App (App (Var (f), Num (11)), Num (2)))))
s => 42
d => 44
l => 42
```

Figure 4: Semantics Example C

D. In this next example, we set x equal to 2 in a function f where we take a y and output the sum of x and y, then setting x equal to 8 and calling f on x.

In the dynamic semantics, when we set x equal to 8, we override the x in the body of the function. Therefore, when we call f on x and therefore set y equal to x, we return $8+8$ or 16. In lexical and substitution semantics, however, we maintain the original binding of x to 2, therefore returning $2+8$ or 10.

```
<== let x = 2 in let f = fun y -> x + y in let x = 8 in f x;;
--> Let (x, Num (2), Let (f, Fun (y, Binop (Plus, Var (x), Var (y))), Let (x, Num (8), App (Var (f), Var (x)))))
s => 10
d => 16
l => 10
```

Figure 5: Semantics Example D1

If we never set x equal to 8, however, then all three semantics will return the same result, as they have the same values for x.

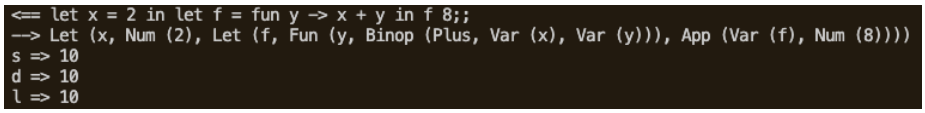

Figure 6: Semantics Example D2

E. In this last example, we set x equal to 1 in a function f where we take a y and output the sum of x and y, then setting x equal to 2 and calling f on 3.

   In the dynamic semantics, when we set x equal to 2, we override the x in the body of the function. Therefore, when we call f on 3 and therefore set y equal to 3, we return $2+3$ or 5. In lexical and substitution semantics, however, we maintain the original binding of 1 to 2, therefore returning $1+4$ or 4.

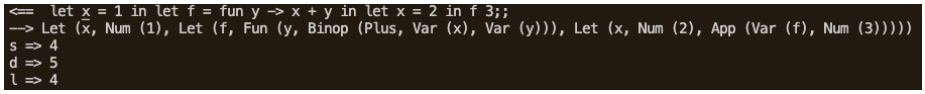

Figure 7: Semantics Example E

3 Float

I added the atomic type of *float* to the language. I did this by updating the type of *expr* to include *Float*, augmenting all *binop* and *unop* evaluators to account for float arithmetic, and extending the MiniML lexer and parser to account for floats.

I decided that the lexer and parser would take symbols of $+$ $-$ $\sim-$ $*$ $/$ for both integers and floats, and that the evaluator would then convert those to $+.$ $-.$ $\sim-.$ $*.$ $/.$ if they were attached to a float. I did this for ease of use for the user.
I raise an error if the user attempts to add, subtract, multiply, or divide different types. Here is an example of my repl evaluating a float:

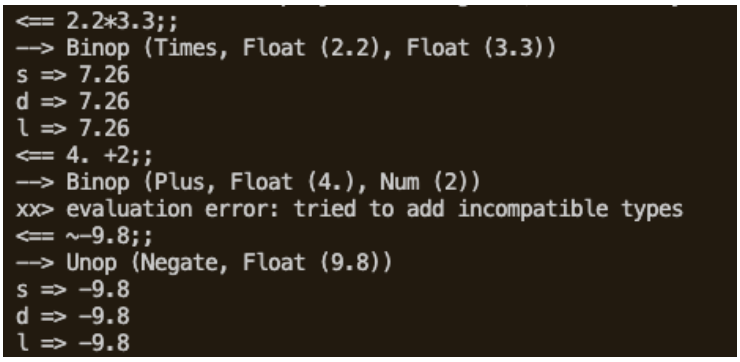

Figure 8: Float Examples

4 Greater Than

I added the symbol > to the language. I did this by updating the type of *binop* to include *GreaterThan*, augmenting all *binop* evaluators to account for float arithmetic, and extending the MiniML lexer and parser to understand > as greater than.

I raise an error if the user decides to compare an integer to a float or vice versa. Here is an example of my repl evaluating a GreaterThan:

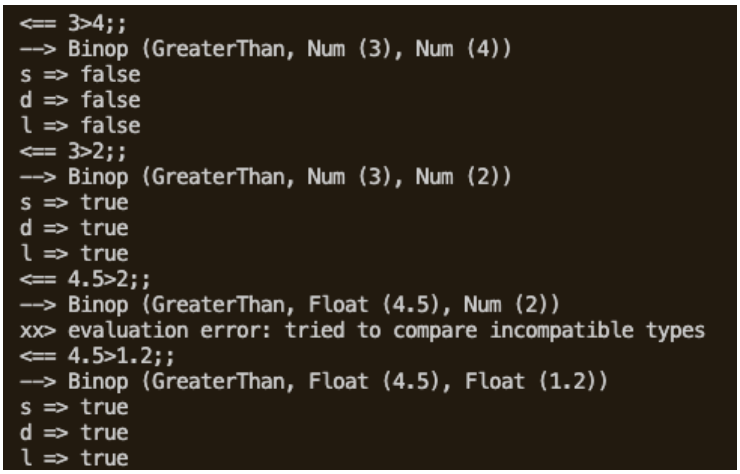

Figure 9: Greater Than Examples

5 Division

I added the symbol / for division to the language. I did this by updating the type of *binop* to include *Divide*, augmenting all *binop* evaluators to account for division, and extending the MiniML lexer and parser to understand / as division.

I raise an error if the user attempts to divide a zero or divide an integer by a float (or vice versa).

Here is an example of my repl evaluating a Divide:

```
<== 9/3;;
--> Binop (Divide, Num (9), Num (3))
s => 3
d => 3
l => 3
<== 9/2;;
--> Binop (Divide, Num (9), Num (2))
s => 4
d => 4
l => 4
<== 8/0;;
--> Binop (Divide, Num (8), Num (0))
xx> evaluation error: divide by zero
```

Figure 10: Integer Division Examples

```
<== 8.5/0;;
--> Binop (Divide, Float (8.5), Num (0))
xx> evaluation error: tried to divide incompatible types
<== 8.5/0.;;
--> Binop (Divide, Float (8.5), Float (0.))
xx> evaluation error: divide by zero
<== 8.6/2.2;;
--> Binop (Divide, Float (8.6), Float (2.2))
s => 3.90909090909
d => 3.90909090909
l => 3.90909090909
```

Figure 11: Float Division Examples