CS51 Final Project Writeup

By Luke Richey

1. Introduction and Overview

The final project of CS51 is a miniature OCaml-like language and interpreter. The project entailed creating expressions, editing a parser, and defining evaluation under different paradigms. In this document, I will be discussing the extensions that I added to the project. These extensions were a lexical environment evaluator, floats and float operations, and a string type that allowed for concatenation.

2. Extension One: Lexical Environment Evaluation

For my first extension, I implemented a lexically scoped environment function to evaluate the various expressions of my MiniML. Now, what makes lexically scoped evaluation different from substitution or dynamic evaluation? The difference between the lexical implementation and substitution is that, in substitution, the language first determines the free variables in the expression to be evaluated, then _immediately _substitutes the values that these variables have been bound to (that is, if they are bound in the first place) until all bindings have been implemented. Lexical utilizes an environment, meaning that these bindings are not substituted as they are under substitution semantics, instead these bindings are stored into the environment where they are substituted for variables when needed (sort of like lazy evaluation). Dynamic evaluation is like lexical, only these bindings in the environment can be changed by later definitions of the variable.

To implement the lexical evaluator, I had to implement the various rules of lexical semantics. This means implementing Closures, which stores the value of the expression and the environment in which it was defined. We need these so that we can later find these bindings when evaluating the expressions. Closures are mostly used when evaluating functions, because we need to maintain the environment and bindings of the expression when it is evaluated. I did not have too much difficulty implementing this, as it was very similar to the other evaluation functions I had made. It was interesting though to think through something like recursion in an environment evaluation setting. Ultimately, lexical rarely evaluates to different values than in the other semantics, except in a case like this:

```
let x = 5 in
let f = fun y -> x + y in
let x = 3 in
f 4 ;;
```

Here, under lexically scoped semantics this would evaluate to 9, because the x in f is substituted for the initial binding to 5 in the first line. However, in a dynamic environment this binding can be changed and x becomes 3, making the expression evaluate to 7 instead. Here are some examples from the MiniMl of the lexical evaluator in action including the one above:

```
<== let rec fact =
        fun n ->
        if n = 0 then 1
        else n * fact (n - 1)
in fact 3 ;;
==> Num (6)
```

```
<== let x = 5 in
let f = fun y -> x + y in
let x = 3 in
f 4 ;;
==> Num (9)
```

3. Extension Two: Floats

For my second extension, I added floats to the language. I also added accompanying float operations, like multiplication, addition, and subtraction. Implementing this into the language itself was not too difficult, it mostly entailed adding a case for floats into my binary operator evaluation function, as well as extending my expressions and binary operators to include the necessary float operations/type. The most challenging part was definitely figuring out how to edit the parser to allow for float and float operator detection in my interpreter. After reading some online sources, I learned some regex (as in regular expressions) that allowed me to implement them. I also had to add additional tokens for the float type itself, the various float operators, and include a rule for the token so that the interpreter knew what to do with it once it encountered a float. Images of this are included and explained in more detail in section 3, but here are some examples of the evaluation (along with some error messages!):

```
<== 4. -. 5. ;;
==> Float (-1.)
<== 5. + 4. ;;
xx> evaluation error: Expected Num
<== 5 +. 4 ;;
xx> evaluation error: Expected Float
<== 5. *. 4. ;;
==> Float (20.)
<== 5. +. 4. ;;
==> Float (9.)
```

4. Extension Three: Strings

My final extension was implementing a string type and an accompanying concatenation operation. This was very challenging, because the regex to recognize strings is pretty complicated. Frankly, I still do not fully understand it but the following regex allows for strings (denoted by quotation marks on either side and any character(s) in between them):

```
let string = ['"']  [^ '"' '\\']+ ['"']
```

This allowed for my parser to detect strings entered into my interpreter and then perform normal operations with them (so long as they were allowed). The most tricky part was figuring out concatenation. Implementing it into the language itself was pretty simple and very similar to the floats implementation, where you sort of copy the other implementations and adjust their types to suit the type you have created. The parser is where the difficulty came in, as screenshots of my implementation are shown below:

```
  let sym_table =
    create_hashtable 8 [
                       ("^", CONCAT);
                       ("=", EQUALS);
                       ("<", LESSTHAN);
                       (">", LESSTHAN);
                       (".", DOT);
                       ("->", DOT);
                       (";;", EOF);
                       ("~-", NEG);
                       ("+", PLUS);
                       ("+.", FPLUS);
                       ("-", MINUS);
                       ("-.", FMINUS);
                       ("*", TIMES);
                       ("*.", FTIMES);
                       ("(", OPEN);
                       (")", CLOSE)
                     ]

let digit = ['0'-'9']
let float = digit* '.' digit*?
let id = ['a'-'z'] ['a'-'z' '0'-'9']*
let string = ['"']  [^ '"' '\\']+ ['"']
let sym = ['(' ')'] | (['$' '&' '*' '+' '-' '/' '=' '<' '>' '^'
                        '.' '~' ';' '!' '?' '%' ':' '#']+)
```

```
 %token CONCAT
%token PLUS MINUS FMINUS FPLUS
%token TIMES FTIMES
%token LESSTHAN GREATERTHAN EQUALS
%token IF THEN ELSE
%token FUNCTION
%token RAISE
%token <string> ID
%token <string> STRING
%token <int> INT
%token <float> FLOAT
%token TRUE FALSE

%nonassoc IF
%left LESSTHAN EQUALS GREATERTHAN
%left PLUS MINUS FPLUS FMINUS CONCAT
%left TIMES FTIMES
%nonassoc NEG
```

In the second image, there are the new tokens I added like concatenation, float operations (denoted by F{operation}), and my string type. I also added the accompanying operators into my sym_table (first image), that way the parser knew what these meant when they were encountered in the interpreter. The regex is shown in the first image as well, towards the bottom. Essentially, the regex for float states that a float is any digit, zero through 9, with a dot attached, as well as an optional digit at the end to allow for floats like 1. instead of 1.0. The final part to add my extensions was to add rules for these new types, as shown below:

```
rule token = parse
  | string as s
        { let new_s = String.split_on_char '"' s in
          let final_s = List.nth new_s 1 in
          STRING final_s
        }
  | float as fnum
        { let num = float_of_string fnum in
          FLOAT num
        }
```

Float is simple, as it just returns the float version of the string wrapped in the FLOAT token. For strings, to allow for concatenation I needed to remove any quotation marks that may be in between a concatenated string (think “CS””51” vs. “CS51”). To do this, I used String.split_on_char which returns a list of all substrings split at the occurrence of a char (in this case, quotation marks). I then obtained the appropriate string from the list (it is always the second element in the list) using List.nth and wrapped this value in my STRING token.

5. Conclusions

Ultimately, I felt this project has taught me a lot about debugging, project management, and the need for sound software development practices. Without good design and style habits, it makes implementing a project like this much more difficult. I realized this as I was implementing this project, as I was constantly thanking my past self for making really good unit tests and pushing my code often. I also learned quite a bit about parsers, regular expressions, and just how a programming language may be implemented.