CS 51 Final Project: MiniML

Jessica Li

3 May 2023

Table of Contents

1 Introduction

For my final project, I implemented new atomic types, unary operators, binary operators, and a lexically-scoped evaluator. In this writeup, I will explain my design choices and implementation for each extension in addition to showing its functionality.

2 Floats, Strings, and Unit

The first extension I implemented was three new atomic types: floats, strings, and unit.

Floats

To implement floats, I changed miniml_lex.mll to include the following definitions of floats: let float = digit+ '.' digit* and

```
| float as ifloat
    { let f = float_of_string ifloat in
      FLOAT f
    }
```

I also edited miniml_parse.mly to include a float token.

Functionally, I added the corresponding operators for floats to have basic arithmetic properties such as negation, addition, subtraction, multiplication, and division that I will expand on in later sections.

Strings

To implement strings, I changed miniml_lex.mll to include the following definitions of strings:

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

```
| string as istring
      { let len = String.length istring in
        let res = String.sub istring 1 (len - 2) in
        STRING res
      }
```

Unit

The last type that I added was the Unit type which is represented by the symbol '()'. As I did with Floats and Strings, I added a unit token to the parser and changed the necessary code in miniml_lex.mll. The Unit type allows functions without inputs to be defined and used in my compiler.

3 Unary Operators

Additonal unary operators that I added include FloatNegate, Sin, Cos, and Tan.

FloatNegate

As an extension to the existing Negate operator for ints, I included the FloatNegate operator to negate floats. In order to do this, I added the symbol ' -.' to miniml_lex.mll and implemented the following code inside my unop_helper function.

```
| Negate, Num n -> Num ~-n
| Negate, _ -> raise (EvalError "can only negate integers")
| FloatNegate, Float f -> Float ~-.f
| FloatNegate, _ -> raise (EvalError "can only negate floats")
```

Sin, Cos, and Tan

New unary operators I implemented were the trig functions: sin, cos, and tan. While these functions normally only work on floats in OCaml, I also wanted to allow these functions to take in ints as they normally can in mathematics. Hence, my implementation involves both floats and changing ints to floats before calling these trig functions.

```
| Sin, Num n -> Float (sin (float_of_int n))
| Sin, Float f -> Float (sin f)
| Sin, _ -> raise (EvalError "can only find sin of integers/floats")
| Cos, Num n -> Float (cos (float_of_int n))
| Cos, Float f -> Float (cos f)
| Cos, _ -> raise (EvalError "can only find cos of integers/floats")
| Tan, Num n -> Float (tan (float_of_int n))
| Tan, Float f -> Float (tan f)
| Tan, _ -> raise (EvalError "can only find tan of integers/floats"))
```

4 Binary Operators

Additional binary operators I added include Divide, Power, and GreaterThan in addition to corresponding operators for floats.

Divide

I implemented the Divide operator which works with two ints by adding the symbol '/" to miniml_lex.mll and editing the parser as needed. We can also see in the photo below my FloatPlus, FloatMinus, FloatTimes, and FloatDivide operators so that floats have the same basic functionality as ints.

Power

Once again, the Power operator was implemented by adding the symbol '**' to miniml_lex.mll and editing the parser as needed. Similarly to Sin, Cos, and Tan, Power normally only works with floats. However, in my implementation, I wanted to add the ability to take powers with ints. Hence, I modified my binop_helper function to change ints to floats before evaluating. One interesting to note here is that I changed the result back into an int as an int raised to an int will always result in an int mathematically.

GreaterThan

The last binary operator I implemented was GreaterThan which is able to compare both ints and floats. This completes the set of compare functions in this compiler.

```
| Plus, Num n1, Num n2 -> Num (n1 + n2)
| Plus, _, _ -> raise (EvalError "can only add integers")
| FloatPlus, Float f1, Float f2 -> Float (f1 +. f2)
| FloatPlus, _, _ -> raise (EvalError "can only add floats")
| Minus, Num n1, Num n2 -> Num (n1 - n2)
| Minus, _, _ -> raise (EvalError "can only subtract integers")
| FloatMinus, Float f1, Float f2 -> Float (f1 -. f2)
| FloatMinus, _, _ -> raise (EvalError "can only subtract floats")
| Times, Num n1, Num n2 -> Num (n1 * n2)
| Times, _, _ -> raise (EvalError "can only multiply integers")
| FloatTimes, Float f1, Float f2 -> Float (f1 *. f2)
| FloatTimes, _, _ -> raise (EvalError "can only multiply floats")
| Divide, Num n1, Num n2 -> Num (n1 / n2)
| Divide, _, _ -> raise (EvalError "can only divide integers")
| FloatDivide, Float f1, Float f2 -> Float (f1 /. f2)
| FloatDivide, _, _ -> raise (EvalError "can only divide floats")
```

```
| Power, Num n1, Num n2 ->
  Num (int_of_float ((float_of_int n1) ** (float_of_int n2)))
| Power, Float f1, Float f2 -> Float (f1 /. f2)
| Power, _, _ -> raise (EvalError "can only take power of integers/floats")
```

5 A Lexically-Scoped Evaluator

The last extension I implemented was a lexically-scoped evaluator. As the readme discussed, the difference between a lexically-scoped evaluator and a dynamically-scoped evaluator is that functions will be evaluated in the environment in which it was defined rather than the environment in which it was called. Hence, eval_l is the same as eval_d except for types Fun, Letrec, and App. The implementation of eval_l involves placing the function within a closure containing its environment and modifying the f inside the environment to have this function as a value.

Below, we can see two examples of the difference between dynamic and lexical environments. The first expression – let x = 1 in let f = fun y ⇒ x + y in let x = 2 in f 3 – evaluates

```
| Equals, x1, x2 -> Bool (x1 = x2)
| LessThan, Num n1, Num n2 -> Bool (n1 < n2)
| LessThan, Float f1, Float f2 -> Bool (f1 < f2)
| LessThan, _, _ -> raise (EvalError "can only compare integers/floats")
| GreaterThan, Num n1, Num n2 -> Bool (n1 > n2)
| GreaterThan, Float f1, Float f2 -> Bool (f1 > f2)
| GreaterThan, _, _ -> raise (EvalError "can only compare integers/floats"))
```

to 5 in a dynamic environment and 4 in a lexical environment. Similarly using float types, the second expression – let x = 1. in let f = fun y ⇒ fun z ⇒ z +. (x /. y) in let y = 2. in f 1. 5. – evaluates to 5.5 in a dynamic environment and 6. in a lexical environment.

```
<== let x = 1 in let f= fun y -> x+y in let x =2 in f 3;;
s==> 4
d==> 5
l==> 4
<== let x = 1. in let f = fun y -> fun z -> z +. (x /. y) in let y = 2. in f 1. 5. ;;
s==> 6.
d==> 5.5
l==> 6.
```