MiniML Evaluator Implementation and Extensions

For the final project in CS51, I completed the MiniML evaluator using the substitution model and dynamic model of evaluation. Alongside the basic requirements, I implemented functionality for strings, floats, basic curried functions, and error-handling in the form of try / with statements. In this writeup, I will go over the four extensions I added, starting with strings.

Strings

To add string support, I first created a new expression type, in expr.ml, String of string. To add this to the parser, I used the regexp expression described in the parsers documentation so that when a user types two quotes with any characters in between, the program recognizes the characters between the quotes as a string. The program cuts off the quotations and stores the user's string, and I implemented "concat" and "length" operations for strings afterwards. Concat, binary operation, can be used with normal OCaml syntax:

```
<== "hello, " ^ "world" ;;
==> String(hello, world)
```

The length operation is a unary operation that uses a new symbol, the hashtag '#'. The length op works as normal in OCaml, returning an int:

```
<== # "hello"
==> Num(6)
```

These operations can combine to create complex functions using strings:

```
<== let rec f x = if (# x) = 6 then x else f (x ^ x) in f "hey" ;;
==> String(heyhey)
```

Floats

Floats are implemented in the parser by accepting any two numbers joined by a period. Their functionality is similar to in OCaml, except the operators used by them are the normal integer operates ('+') instead of OCaml float operators ('+.'). However, this is not a problem because the interpreter will still not allow binary operations on different types. The floats can be used as follows:

```
<== 2.0 + 1.5 ;;
==> Float(3.5)
```

Curried Functions

My version of MiniML supports curried "let" and "let rec" functions with one argument at the moment. This was done mostly in the parser file as so:

```
| LET ID ID EQUALS exp IN exp         { Let($2, (Fun($3, $5)), $7) }
| LET REC ID ID EQUALS exp IN exp     { Letrec($3, (Fun($4, $6)), $8) }
```

This code simply reformats the typed expression to be interpreted as if it was not curried by the evaluator. With this, users can more easily type and visualize more complex functions, such as the recursive countdown:

```
<== let rec f x = if x = 1 then true else f (x - 1) in f 100 ;;
==> Bool(true)
```

Try / With Statements

I implements try / with statements first by creating a new expression type, Try of expr * expr. The parser uses new keywords "try" and "with" to see when the user is using a try / with statement. The evaluator returns the second expression only if the first statement returns any kind of exception. I decided to make the exception anonymous since within the evaluator, the only exception that I raise is EvalError, so it would not make sense to match the second statement's return against specific exceptions.

```
<== try let f x = x ^ x in f 3 with 100 ;;
==> Num(100)
```