CS51 Final Project Extension Writeup

By: Gary Wu

Table of Contents

1. Types

My final project is extended with two additional Atomic types: Floats and Strings.

Floats

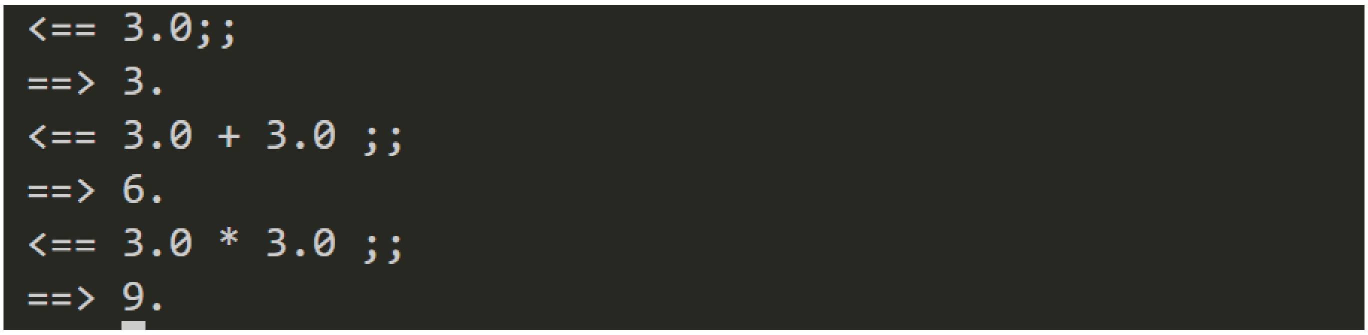
For floats, I decided to edit the miniml_lex.mll file with an additional float type defined as follows:

```
let float = digit* '.' digit*

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

I change the parser file - miniml_parse.mly - file with the following code:

```
%token <float> FLOAT

| FLOAT                 { Float $1 }
```

Floats are treated almost identically to Nums as currently implemented by the given code, with the motivation being to maintain consistency with what most users are expecting the application to perform. Thus, an expression like "Float(2.0)" is evaluated almost exactly how something like "Num(3)" would be, throughout each of the evaluator functions. The expr.ml and expr.mli files are edited accordingly to account for the addition of Float expressions. They are also treated in similar capacities to Nums in the "expr.ml" file. Binary operators for floats exist in the same capacity as Nums, appropriately accounted for in the expr.ml file and most importantly in the evaluation.ml file with an added match case if both inputs to the Binop are floats. For the purposes of consistency, if the user inputs something like "3.0 * 3", an error will be returned.



```
<== 3.0;;
==> 3.
<== 3.0 + 3.0 ;;
==> 6.
<== 3.0 * 3.0 ;;
==> 9.
```

Strings

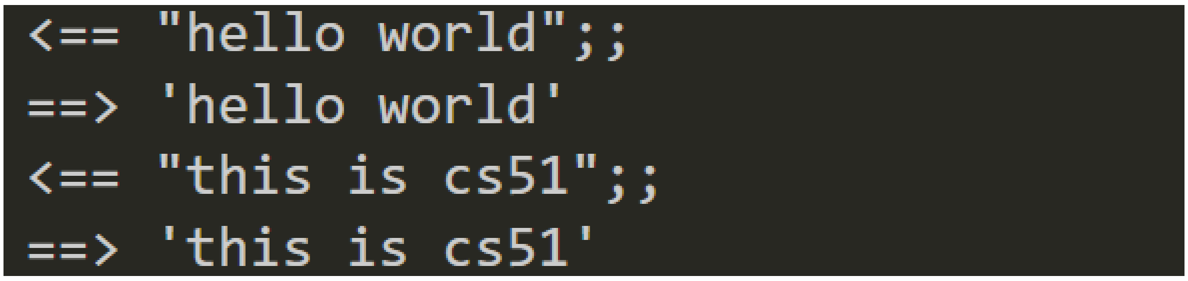
A similar philosophy is applied to the second extended Atomic type : strings. First, the .mll file must be edited with the regex such that the content inside quotation marks are considered strings, given that there aren't any nested quotation marks.

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

| string+ as istring
    { let string = String.concat "" (String.split_on_char '"' istring) in
      STRING string }
```

I choose the String.split_on_char function to remove any single quotation marks from the istring and then concat the string list with an empty string (this is relevant for the eventual concat operator). The extra workings for the parser file are also implemented:

```
%token <string> STRING

| STRING                { String $1 }
```

To help distinguish strings from other alphabetical expressions, I decided to add single quotes to the output of string expressions on their own, as seen in exp_to_concrete_string.



```
<== "hello world";;
==> 'hello world'
<== "this is cs51";;
==> 'this is cs51'
```

Similar to Floats, the evaluators treat strings almost exactly like they do with other atomic types, with the inclusion of a additional binop expression: the Concat operator, discussed in section 3.

2. Unary Operators

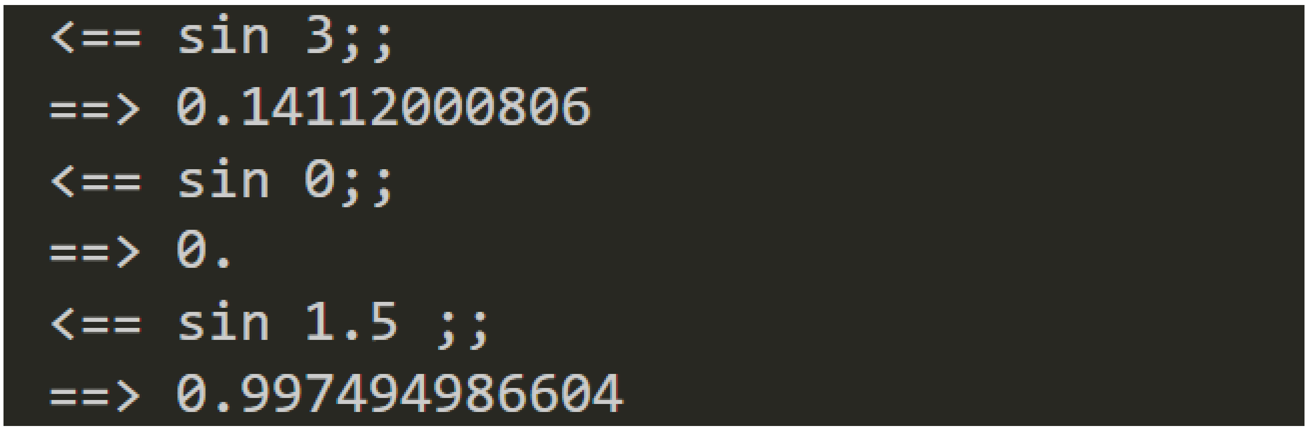
Sin Operator

The Sin Operator accepts Num and Float expressions and outputs the correspong Sine of that value in the form of a float regardless of whether the input was a Num or Float. This is done in consideration of the user, as Sin calculations are often done with the need of precision, and it wouldn't make sense to output whole numbers in this scenario. An error is returned if the types of the inputs don't match, or aren't nums and floats.

Implementation of this operator including creating a keyword "sin" in the .mll file, as it there isn't a clear symbol that would correspond to a sin calculation. The parse file includes a SIN token, attribution to non-associative operator, and this grammar:

```
| SIN exp                 { Unop(Sin, $2) }
```

Evaluation is pictured here and shows that regardless of num or float input type, the expression will evaluate to a float. Thus, one of the benefits of the float implementation is added precision in calculations, seen in Sin calculations.



```
<== sin 3;;
==> 0.14112000806
<== sin 0;;
==> 0.
<== sin 1.5 ;;
==> 0.997494986604
```

Abs Operator

The Abs Operator also accepts Num and Float expressions, and outputs the absolute value of the expression, but keeps the initial type: abs(Num(x)) would return an integer while abs(Float(f)) would return a float. This is done for consistency across evaluations of expressions, and the less important effect of precision. The implementation is similar to Sin, but uses a symbol rather than keyword -> "~+".

3. Binary Operators

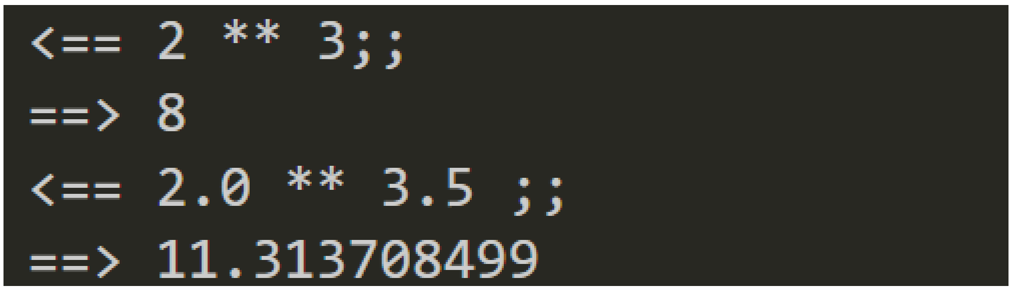
Power Operator

The power operator takes in two numbers (nums and floats) and outputs the first input to the power of the second input. This operator is implemented for both nums and floats, as float powers can give more precision if needed for the user.

A ** operator is included in the symbol table in the .mll file and is implemented as follows in the parse file:

```
%token POWER

| exp POWER exp         { Binop(Power, $1, $3) }
```

A demonstration of evaluation is below:

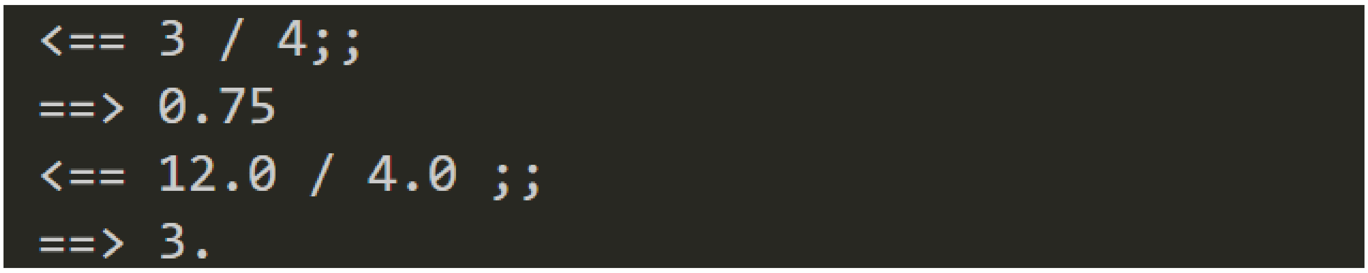


```
<== 2 ** 3;;
==> 8
<== 2.0 ** 3.5 ;;
==> 11.313708499
```

Divide

The divide operator outputs the value of the first input divided by the second input. Notably, the expression returns a float regardless of input types, and this is done to preserve precision calculations - an added benefit to the implementation of float types. The same formula for the parser and lex files remains, so will not be repeated here.

Demonstration of Divide:

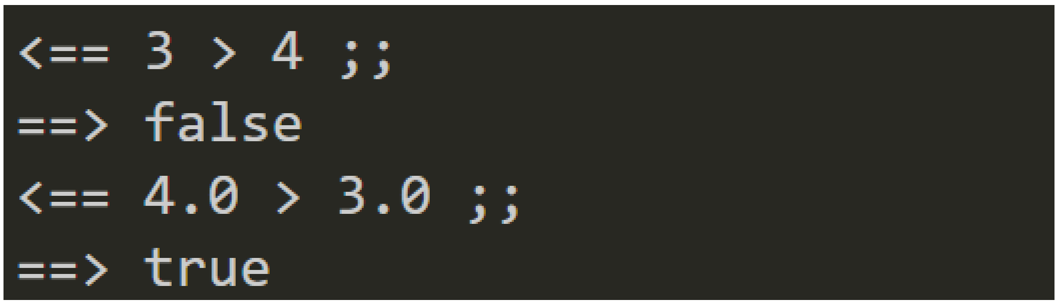


```
<== 3 / 4;;
==> 0.75
<== 12.0 / 4.0 ;;
==> 3.
```

GreaterThan

The GreaterThan operator functions exactly as the LessThan operator does, just returning the opposite. The same formula for changing the parser, lex, and expr files applies for this operator, so will not be repeated here.

Demonstration of GreaterThan:



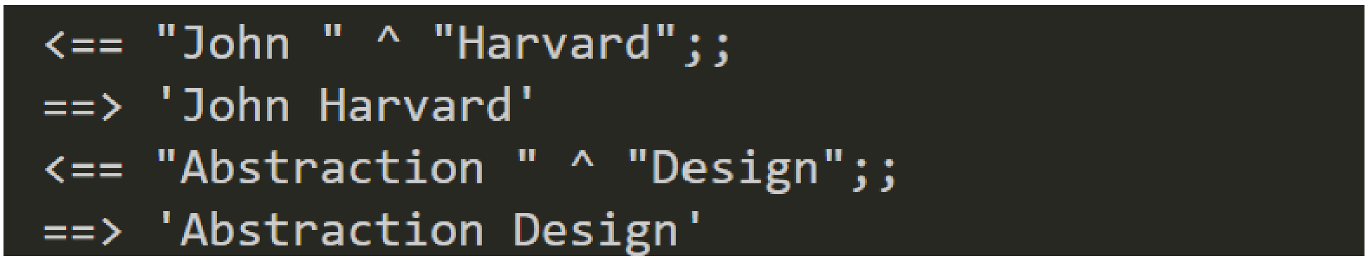
```
<== 3 > 4 ;;
==> false
<== 4.0 > 3.0 ;;
==> true
```

Concat

The addition of the String type allows for the implementation of a string concatenation operator, which takes two strings and combines them. Crucially, the String.split_on_char function used to define strings in the .mll file doesn't allow quotation marks in strings, which is useful for the concat operator, which would have outputted something like "cs""fifty" without such implementation. This operator only works on two strings and raises an EvalError if otherwise. Here is the parser file code:

```
%token CONCAT

| exp CONCAT exp        { Binop(Concat, $1, $3) }
```

Demonstration of Concat:



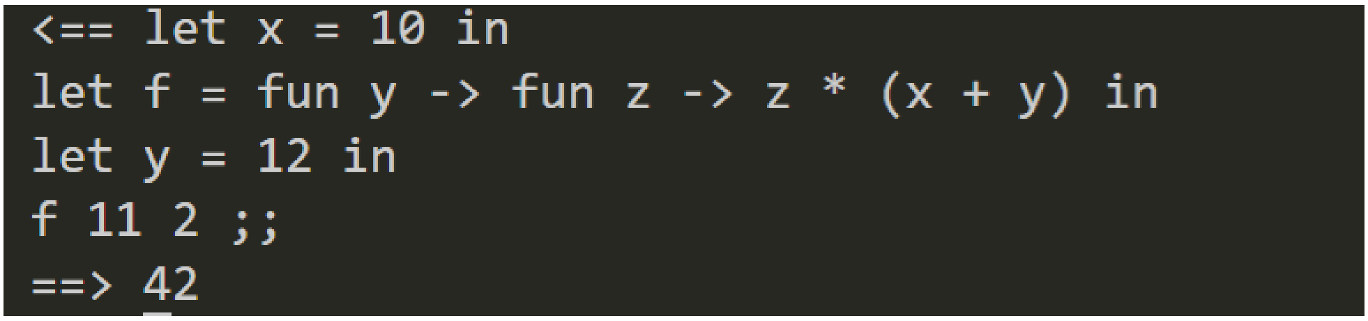
```
<== "John " ^ "Harvard";;
==> 'John Harvard'
<== "Abstraction " ^ "Design";;
==> 'Abstraction Design'
```

4. Lexical Evaluator

Finally, a Lexical Evaluator is also implemented. Following the advice and rules from the textbook, the lexical evaluator is similar to the dynamic evaluator in all areas except for Fun, Letrec, and App. Thus, the implementations of these functions differ from eval_d. For function, lexical returns a closure with the environment rather than just itself. For letrec, we follow the rules on page 412 of the textbook, in which we implement an environment with a mutable store.

We also know that eval_l and eval_s should provide similar outputs which is helpful in testing and comparison to eval_d.

Demonstration of Lexical Evaluator (evaluating to 42 instead of 44 in dynamic):



```
<== let x = 10 in
let f = fun y -> fun z -> z * (x + y) in
let y = 12 in
f 11 2 ;;
==> 42
```