CS51 Final Project

Brad Campbell

bfcampbell@college.harvard.edu

May 2021

1 Introduction

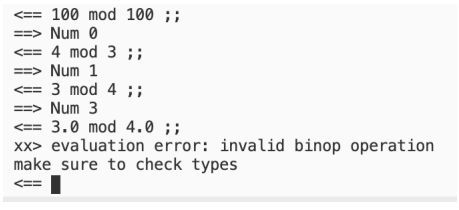
The following write-up details different extensions I implemented in the CS51 Final Project of the Spring 2021 term. They are in number order.

2 Divide

For the first extension of my project, I decided to add divide capabilities. I thought this was appropriate given that there was already a multiplication method.
The functionality of divide was accomplished with additions in *expr.ml*, *expr.mli*, *miniml_lex.mll*, *miniml_parse.mly*, and *evaluation.ml*. In pursuit of brevity, I have decided not to detail these changes in this report. They can be found in each respective file.

3 Modulo

I thought it might be interesting to add a modulo operation, especially since this is extremely important in many algorithms. I decided to reserve the word "mod" just as many languages, including OCaml, do. The process of adding this operation was similar to adding the Divide operator. Its functionality is demonstrated with this screenshot:



4 Increased Boolean Support

Along with the Equals and LessThan operators that were already in place, I added LessThanOrEqual, GreaterThan, GreaterThanOrEqual, And, Or, ExclusiveOr, and Not operators to expand the capabilities of the user. These were implemented with the same operator symbols that OCaml uses (<= , > , >= , && , || , <> , not). The following screenshots demonstrate the usage of each new operator.

```
<== 42.42 <= 42.42 ;;
==> Bool true
<== 42.42 <= 84.84 ;;
==> Bool true
<== 84.84 <= 42.42 ;;
==> Bool false
<== 84 > 42 ;;
==> Bool true
<== 42 > 84 ;;
==> Bool false
<== 42.42 >= 42.42 ;;
==> Bool true
<== 42.42 >= 42.43 ;;
==> Bool false
<== not true ;;
==> Bool false
<== not false ;;
==> Bool true
<== not not true ;;
==> Bool true
<==
```

```
<== true && true ;;
==> Bool true
<== true && false ;;
==> Bool false
<== false && false ;;
==> Bool false
<== true || true ;;
==> Bool true
<== true || false ;;
==> Bool true
<== false || false ;;
==> Bool false
<== true <> true ;;
==> Bool false
<== true <> false ;;
==> Bool true
<== false <> false ;;
==> Bool false
<==
```

5 Float

With my expanded operations working, I thought float capabilities could be added next. This was implemented with code changes similar to the Divide functionality.

My implementation makes the "." optional in float operations. I thought this was a tedious feature of OCaml, so my project does not require the series of $+., -., *., /.$ operators. It does, however, appropriately recognize these operators for users who still wish to use them. Similar to OCaml, if a float is mixed with an integer, my implementation will throw an error. The screenshot below demonstrates some Float usage:

```
<== 3. +. 4. ;;
==> Float 7.
<== 3. + 4. ;;
==> Float 7.
<== 3 + 4 ;;
==> Num 7
<== 3. + 4 ;;
xx> evaluation error: invalid binop operation
make sure to check types
<==
```

6 String

Using the same code changes as the Float implementation, I made a String type available for use. I also added a concatenation feature to allow strings to be joined. This was done with the standard OCaml carrot (^) operator. The String implementation can be shown with the following screenshot:

```
<== "hello, world!" ;;
==> String "hello, world!"
<== "hello" ^ " again" ^ ", world!" ;;
==> String "hello again, world!"
<==
```

7 Hexadecimal Numbers

I decided to implement hexadecimal support. This relies on the prefix "0x" followed by a string of characters that are 0-9 and/or A-F (I also supported a-f because hexadecimals can be found in both uppercase and lowercase formats). This implementation was relatively straight forward because OCaml provides support for converting hexadecimal numbers into integers. Because of this feature, I decided to perform operations with the already-existing Num type. The usage of hexadecimal numbers is shown with this screenshot:

```
<== 0xFA ;;
==> Num 250
<== 0xfa ;;
==> Num 250
<== 0xFA * 0xfa ;;
==> Num 62500
<==
```

8 Evaluator Function Structure

Before detailing my project's approach to the lexical extension, I wanted to explain the structure of my various eval functions. I created a datatype called *model* that stores the possible evaluation semantics of the project. I then made a reference variable, called *curr_mod*, that represents the current model the program will run. This is shown with this code snippet:

```
type model = Substitution | Dynamic | Lexical ;;

let curr_mod = ref Substitution ;;
```

I then grouped all of my eval functions into a singular universal function called *eval_uni* that evaluates an expresssion according to the value that is stored at the *curr_mod* variable. To ensure that the CS51 test cases ran appropriately, I maintained the argument types as shown with this function declaration snippet:

```
let rec eval_uni (exp : expr) (env : Env.env) : Env.value =
    ... ;;
```

Taking advantage of the *model* type and *curr_mod* reference variable, each type of semantic approach was easily deployable. This is demonstrated with the following code snippet of *eval_s* (the *eval_d* and *eval_l* are implemented similarly):

```
let eval_s (exp : expr) (env : Env.env) : Env.value =
    curr_mod := Substitution;
    eval_uni exp env ;;
```

I decided on this approach because it meant one less parameter for the *eval_uni* function to tediously pass itself in recursive calls.

9 Lexical Evaluation

Early on, I learned that the only difference between dynamic and lexical would be the Fun and App segments. This made the universal evaluator function even more viable since the lexical and dynamic approaches would have much of the same code.

For the Fun expressions, the lexical approach needed to have a closure that marked a specific environment from compile-time. In contrast, the Fun expressions in the dynamic approach relied on the run-time environment.

For App expressions, the first step is evaluating the first term. Because only a function can be applied, this should always be a function (I raise an EvalError if this is not the case). As detailed above, the lexical approach will yield a closure for a function evaluation. Therefore, for the lexical approach, I matched the Env.Closure and then made a reference to the evaluation of the second term of the App expression. Finally, I returned the evaluation of the consequence of the Env.Closure Fun expression according to the environment at the moment in time when the closure was evaluated. This step is what fundamentally separates the dynamic and lexical approaches.

10 User-Friendly Extension

I thought about user-friendliness and ended up writing a section of code that would allow a user to choose the semantics approach they wanted at run-time. I decided not to include this piece of code in the *evaluation.ml* file so that the CS51 tester would run correctly. I created a file called *alternative_evaluation.ml* to store this alternative approach in case its reference is useful. The following

screenshot shows the user experience of running this code:

```
bradcampbell@brads-macbook project % ocamlbuild -use-ocamlfind miniml.byte
Finished, 18 targets (15 cached) in 00:00:00.
bradcampbell@brads-macbook project % ./miniml.byte


Please enter a valid semantics model and press enter:
"s" for substitution, "d" for dynamic, "l" for lexical.
(invalid inputs will result in Substitution being chosen)
d

DYNAMIC MODEL

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

11 Testing Approach

Coming into this project, my goal was to write tests that were extremely thorough. I found many small errors from this approach and it proved useful in helping me build confidence in the extensions I wrote. My tests can be found in the *tests.ml* file.

12 Conclusion

I tried to keep this write-up as short as possible to save your time. I have included screenshots to better explain how each element can be used. All code can be found in the various files I have referenced.

Thank you CS51 Staff for a great semester!