CSCI E-51 Final Project Write-Up

Gabriel Chiong

May 2021

Abstract

This document contains the write-up for my CSCI E-51 Final Project at Harvard University's Extension School. This project involves implementing a Turing-complete, metacircular interpreter for a subset of the OCaml programming language. The final project specifications require implementations of both the SUBSTITUTION and DYNAMIC ENVIRONMENT semantic models for specifying the semantics of the programming language. Furthermore, the specification requires that the student implements extensions to the interpreter, with lexical scoping of the environment model being a highly recommended option.

For my project, I have implemented both the SUBSTITUTION and DYNAMIC ENVIRONMENT models, along with several other extensions to the language: adding additional atomic types and operators, modifying the environment semantics to to manifest lexical scope, augmenting the syntax to allow for the curried function definition syntactic sugar, and including references and mutability to the language.

1 Introduction

The language implemented in this project only includes a subset of OCaml-like constructs. The two main limitations of this language consist of: limited support for types (and no user-defined types), and no type inference (only enforced during run-time). Despite these limitations, the language implemented is Turing-complete, which is as capable as any other programming language according to the Church-Turing thesis. In the interest of brevity, the programming language implemented in this project shall henceforth be known as "MiniML" within this document, which seems appropriate given the minimal size and limitations of the language.

1.1 Project requirements

The main goal of this project is to implement various forms of interpreters that vary in the semantics they manifest. It will be based on expressions formed by the OCaml programming language. This project can be thought of as being divided into three parts. The first two parts are a hard requirement to implement two different interpreters based on the SUBSTITUTION, and DYNAMIC

ENVIRONMENT semantics. Following the completion of these two different interpreter models, the third and final part of the project requires that the language be extended to incorporate various different constructs, of which the student is able to decide which to implement. Within this project, I have decided to implement the following extensions to MiniML:

1. Add additional atomic types - `Float` for floats, `Str` for strings, and the `Unit`, for signifying an expression which returns no value. Associated operators were also included in order to maintain the strong typing characteristic found in OCaml. Extensions were also made to the provided atomic types in the distribution code such as `GreaterThan` as a dual to the provided `LessThan`, as well as boolean operators `And` and `Or` (respectively `&&` and `||` in OCaml).

2. Implementing lexical scoping as an alternative to dynamic scoping for the environment model. This includes defining different evaluation rules for the semantics of the dynamic environment model to more accurately replicate the behavior of the OCaml programming language.

3. Augmenting the interpreter syntax to allow for the "syntactic sugared" version of curried function definition, both for anonymous functions and for named function definitions. This mainly involved modifying of parsing rules to identify instances of "sugared" function definitions.

4. Adding references and mutability functionality - the `Ref`, `Deref`, `Sequence`, and `Assign` operators (respectively `ref`, `!`, `;`, `:=` in OCaml) were added to to allow for a lexically scoped environment model with mutable storage. Evaluation rules for mutable storage were also implemented in order to handle state change.

As the first two parts of the project, SUBSTITUTION and DYNAMIC ENVIRONMENT semantics are implemented based largely on stub code provided in the distribution files, this write-up will not delve into the specifics of how these were implemented. This document will mainly contain the details of implementing the extensions made in the third part of the project.

1.2 Additional functionality

Additional functionalities were added to maintain robust and efficient code, and separate from the hard requirements of this project. These were a byproduct of implementing MiniML, and not part of the required functionalities. They are namely, a comprehensive suite of unit tests and accompanying framework (the `tests.ml` file), and a modification to the REPL (the `miniml.ml` file) to simplify the process of debugging. These are described briefly in the two short subsections below.

1.2.1 Unit testing framework

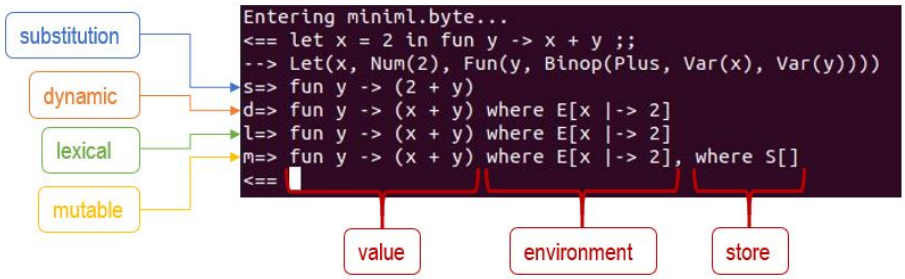
In addition to the implementations as described in the previous section, a comprehensive testing framework and set of unit tests were developed within a file named `tests.ml`, which tests every implemented function thoroughly, while maintaining code re-usability by running the same tests for SUBSTITUTION, DYNAMIC ENVIRONMENT, and LEXICAL ENVIRONMENT, albeit with differing results expected where appropriate. The course's `CS51Utils` module was chosen as the unit testing framework rather than using `assert` statements, as the status messages are more descriptive, hence debugging would be a simpler process using `CS51Utils`, and namely the `unit_test` function from this module.

1.2.2 REPL extensions

As an aid to debugging while writing the code, and subsequently as a way to display the features implemented in this project, several changes were made to the distribution code of the MiniML REPL file `miniml.ml`. A helper function called `print_res` was used to print the results of evaluating different semantic models, in a similar vein to Professor Shieber's version in his demonstration video. For every expression typed into the interpreter, the user is able to see the results of evaluation from different semantic models: SUBSTITUTION, DYNAMIC ENVIRONMENT, LEXICAL ENVIRONMENT, and LEXICAL MUTABLE ENVIRONMENT. Where environment semantics apply, the relevant environment upon expression evaluation is displayed correspondingly. Similarly, where storage and mutable semantics apply, the relevant storage environment upon expression evaluation is displayed.

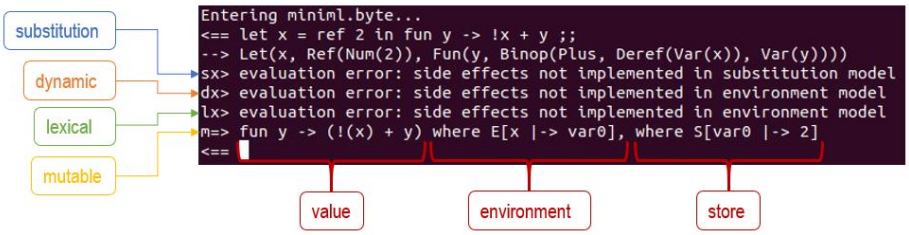
The two images below illustrates the different values printed for each corresponding semantics model.

Figure 1: The REPL display for anonymous function definition where no state change effects are needed.



As observed in Figure 1, where no operators for state change is defined, all four semantics are shown, along with their corresponding environments (if applicable), denoted by $E[...]$, and the store is empty in this case (denoted by $S[...]$). However, in Figure 2, state change is present. The SUBSTITUTION, DYNAMIC ENVIRONMENT, and LEXICAL ENVIRONMENT, are undefined and throws

Figure 2: The REPL display for anonymous function definition where state change is present.



an `EvalError`. The LEXICAL MUTABLE ENVIRONMENT however, handles this by displaying the environment, which maps relevant variables to their locations in the environment $E[...]$ and the location to their value in $S[...]$. The case of anonymous functions was used as an example to highlight the differences between the different REPL values printed, and the same principles apply to other constructs within the language which are affected by environments and stores, for example `Let` statements.

2 Additional atomic types

The first extension made to the MiniML language was adding additional atomic types. I decided to add floats, strings and the unit, along with some of their frequently used operators in native OCaml. As these were new constructs for MiniML, changes to the lexer and parser files were required. In addition to this, the `expr.mli`, `expr.ml`, and `evaluation.ml` files would need to be updated as well. These are described in the subsections below.

2.1 Lexer changes

The lexer for MiniML is written using a program called `ocamellex` and is contained within the file `miniml_lex.mll`. In order to maintain a strong, static typing regime for MiniML similar to native OCaml, additional operators were introduced into the `sym_table` hashtable where needed (such as differentiating between arithmetic operators for ints and floats).

For floats, the symbols `+.`, `-.`, `*.`, and `~-.` were added as a dual to the operators from the given integer type, `Num`. No rule changes were needed for parsing these symbols, as they had already been introduced into the `sym_table` hashtable. However, an additional parsing rule was needed to handle expressions involving float values. It works similarly to what was implemented in the distribution code for `Nums`, and is in effect, a combination of integers with a '.' character within or at the end of the expression.

For implementing the string atomic type, the concatenation symbol `^` was first added to the `sym_table` hashtable, and a definition of strings was added us-

ing regular expressions as `let string = ['"'] [^'"']* ['"']`, which allows any character between the two quotation marks, except for another quotation mark. An additional parsing rule was added for strings, in which I had to remove the quotation marks using OCaml's `Str` module for regular expressions and string manipulation.

The unit was the easiest out of the three additional atomic types to include. It simply consisted of adding the unit symbol, `()` to the `sym_table` hashtable, and a simple parsing rule for whenever that symbol is encountered.

As a minor improvement to comparison operators, I added the "greater than" operator, the `>` symbol as having only a `<` operator did not feel right, intuitively speaking. The boolean operators "and" (`&&`) and "or" (`||`) were also added to the `sym_table`, as well as the `not` keyword to the `keyword_table` hashtable for inverting boolean values.

2.2 Parser changes

The parser for MiniML is written in a program called `menhir` and is contained within the file `miniml_parse.mly`. Tokens for values and their operators, as well as their precedence rules were added accordingly, with care taken to implement an order which is consistent with native OCaml. The grammar of the language was also updated to include the newly introduced atomic types, but these changes are trivially based on the structure of existing grammar rules. These can be best understood by reading through the `miniml_parse.mly` file therefore, this document will not be covering the parser for newly introduced atomic types.

2.3 Expressions and evaluation changes

Within the `expr.mli` and `expr.ml` files, the type definitions for floats, strings and units were added to the existing ones. Similarly, the float operators, comparison operator and boolean operators introduced were added to the type definitions for `binop` and `unop`. `FNegate` for float negation, and `Not` for boolean negation were added to the `unop` type. `FPlus`, `FMinus`, and `FTimes` as float operators, `Concat` as a string concatenation operator, `GreaterThan` as the comparison operator, and `And` and `Or` as the boolean operators were added to the `binop` type definitions.

The float atomic type was introduced as `Float of float`, strings are added as `Str of string`, and units as a `Unit`. As these newly introduced constructs are atomic types, the `free_vars` and `subst` functions within the `expr.ml` file did not require excessive modification. Similarly, functions to produce string representations of both the abstract tree syntax and concrete syntax were trivially updated to include these new atomic types. In the same vein, updating the `evaluation.ml` file was largely a mechanical exercise, with trivial differences to the existing atomic types. Overall, this was perhaps the simplest extension to the project to implement.

3 Lexical scoping

Native OCaml is itself, lexically scoped. This is in contrast to the DYNAMIC ENVIRONMENT implemented in the first part of the assignment. The main difference between the two types of environment semantics is that LEXICAL ENVIRONMENT semantics "capture" the lexical environment at the time of function definition for use when the function is applied. DYNAMIC ENVIRONMENT semantics do not store the lexical environment at function definition, and applies the dynamic environment during function application. LEXICAL ENVIRONMENT semantics are consistent with the result for SUBSTITUTION semantics. The differences are perhaps best illustrated by the following expression.

```
let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 3 ;;
```

In this expression, we would expect a dynamically scoped semantics model to evaluate to a value of 5 as the environment in play during function application contains the mapping of $x \mapsto 2$. However, the SUBSTITUTION and LEXICAL ENVIRONMENT models should return a value of 4, since the lexical environment at the point of function defintion contained the mapping of $x \mapsto 1$. Conceptually, the notion of capturing the lexical environment at the time of function definition for later use during function application can be implemented as a "closure" data structure, which contains an expression and its lexical environment.

Implementation-wise, there are only minor changes to the DYNAMIC SEMANTICS from the first part of the project, with the most significant being the evaluation of `Letrec`. However, adding lexically scoped evaluation rules was largely a mechanical exercise in applying rules from the CSCI E-51 course. In the interest of succinctness, I have defined a new type, `type model = Dynamic | Lexical`, along with a common helper function to carry out evaluation of an expression depending on the type of model passed in as an argument to the function in the file `evaluation.ml`. The heading of the function is as follows:

```
let eval_env (exp : expr) (env : Env.env) (model : model)
           : Env.value =
  ...
```

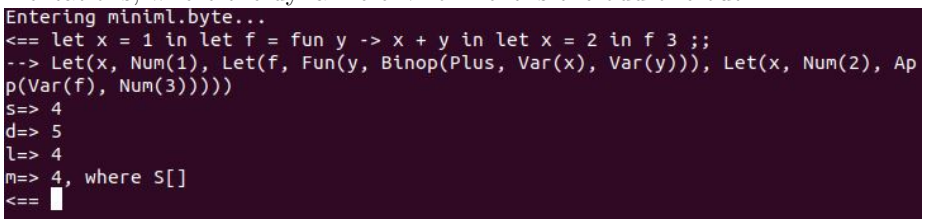
During function application, the `eval_env` function checks the type of the model passed in as an argument, in order to determine whether to manifest the dynamic or lexical environments in evaluating a result. This helper function eliminates significant amounts of repeated code, as the `eval_d` and `eval_l` functions can now be written with a single line of code.

```
(* Dynamically-scoped environment model evaluator *)
let eval_d (exp : expr) (env : Env.env) : Env.value =
  eval_env exp env Dynamic ;;
```

```
(* Lexically-scoped environment model evaluator *)
let eval_l (exp : expr) (env : Env.env) : Env.value =
  eval_env exp env Lexical ;;
```

With these changes added to the `evaluation.ml` file, the interpreter is now able to evaluate expressions in the SUBSTITUTION, DYNAMIC ENVIRONMENT, and LEXICAL ENVIRONMENT model settings, as seen in the image below. The substitution evaluation rules return a result of 4, consistent with the lexical environment rules. As expected, the dynamic environment rules return a result of 5. The mutable environment is also lexically scoped, hence it returns a value of 4, but a more detailed documentation of the environment for mutable storage is provided in the succeeding sections.

Figure 3: Comparison of evaluation results from the different semantics implementations, where the dynamic environment is the odd one out.



4 Syntactic sugar

The third extension to this project is adding syntactic sugar for function definition. Syntactic sugar is additional syntax which serves to abbreviate a more complex construction. Not only does the syntactic sugar implemented in this project's extension for function definition simplify the need for constructs like `let f = fun x -> ...` to `let f x = ...`, but it also simplifies curried function definition (multiple argument functions for a programming language based on Alonzo Church's Lambda Calculus).

```
(* Non-syntactic sugared function definition with currying *)
let f =
  fun x ->
    fun y ->
      fun z ->
        ...

(* Syntactic sugared function definition with currying *)
let f x y q -> ...
```

Likewise, this notion of syntactic sugar for curried function definition also applies to anonymous functions.

```
(* Non-syntactic sugared anonymous function with currying *)
fun x ->
  fun y ->
    fun z ->
      ...

(* Syntactic sugared anonymous function with currying *)
fun x y z -> ...
```

The only change required for this extension is located within the MiniML parser, `miniml_parse.mly`. This was the most challenging extension to implement, although the total number of lines of code needed was perhaps the shortest of all extensions.

Within the grammar rules section, the rules for `LET`, `LET REC`, and `FUN` would need to be changed from `ID`, which is a single variable, to multiple variables in order to account for syntactic sugared curried function definition. A new rule was needed to account for multiple variables, rather than a single variable. This new rule was named as `vars`, to signify that it represented multiple variables. The most basic OCaml construct for composite types, a list, was used to define this new rule. Utilising the concept of forming composite data types by alternation, `vars` could either take the form of a single variable, or more than one variable, from which the parser would interpret as the user wishing to use the sugared version of curried function definition.

With the grammar rules established, the most difficult part of this extension was in determining how to represent a list of variables as their abstract tree syntax form for evaluation. Eventually, this was implemented using helper functions written in OCaml within the header of the parser file, to deconstruct the list - variable by variable, and form a nested abstract tree syntax expression for evaluation. Within the header of the `miniml_parse.mly` file, the two functions `uncurry_fun` and `uncurry_let` fulfill the role of deconstructing the variable list into the relevant abstract tree syntax. Since these functions return an abstract tree syntax expression, the grammar rules for `LET`, `LET REC`, and `FUN` were changed to use these helper functions, rather than manually constructing the abstract tree syntax, as was originally done for single argument functions.

As a simple demonstration, the following concrete syntax can be entered into MiniML's interpreter, with the results shown in the images below:

```
let intofbool b =
  if b then 1
  else 0 in
intofbool true ;;

let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1) in
fact 10 ;;
```

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

let f x y z = x + y * z in
f 2 3 4 ;;
```

Figure 4: Syntactic sugar for single argument functions.

```
Entering miniml.byte...
<== let intofbool b = if b then 1 else 0 in intofbool true ;;
--> Let(intofbool, Fun(b, Conditional(Var(b), Num(1), Num(0))), App(Var(intofboo
l), True))
s=> 1
d=> 1
l=> 1
m=> 1, where S[]
<==
```

Figure 5: Syntactic sugar for single argument recursive functions.

```
Entering miniml.byte...
<== let rec fact n = if n = 0 then 1 else n * fact (n - 1) in fact 10 ;;
--> Letrec(fact, Fun(n, Conditional(Binop(Equals, Var(n), Num(0)), Num(1), Binop
(Times, Var(n), App(Var(fact), Binop(Minus, Var(n), Num(1))))), App(Var(fact),
Num(10)))
s=> 3628800
d=> 3628800
l=> 3628800
m=> 3628800, where S[]
<==
```

Figure 6: Syntactic sugar used together with single argument function definition.

```
Entering miniml.byte...
<== let x = 10 in let f y = fun z -> z * (x + y) in let y = 12 in f 11 2 ;;
--> Let(x, Num(10), Let(f, Fun(y, Fun(z, Binop(Times, Var(z), Binop(Plus, Var(x)
, Var(y))))), Let(y, Num(12), App(App(Var(f), Num(11)), Num(2)))))
s=> 42
d=> 44
l=> 42
m=> 42, where S[]
<==
```

As an aside, Figure 7 shows the dynamic environment throwing an error stating that the y variable is unbound during evaluation. This is the expected behaviour of the DYNAMIC ENVIRONMENT semantics model as the function is returned outside of the environment in which y is defined. In the lexical environment, the y is available but in the dynamic environment, there is no mapping for y. There are various ways to fix this, namely defining a value for the y variable

Figure 7: Syntactic sugar with multiple arguments function definition.

```
Entering miniml.byte...
<== let f x y z = x + y * z in f 2 3 4 ;;
--> Let(f, Fun(x, Fun(y, Fun(z, Binop(Plus, Var(x), Binop(Times, Var(y), Var(z))
)))), App(App(App(Var(f), Num(2)), Num(3)), Num(4)))
s=> 14
dx> evaluation error: unbound variable, y
l=> 14
m=> 14, where S[]
<==
```

prior to function application, however this is outside the scope of the discussion in this section. This is the reason most modern programming languages implement lexical scoping rather than dynamic scope.

Unfortunately, due to time limitations for this project, I was unable to add functionality for no-argument functions, in the vein of `fun () -> ...`, `let f () = ...`, or even `fun _ -> ...`. This is proposed to be the scope of future extensions to make this language more expressive.

5 References and mutability

The final extension implemented in this project is to implement references for mutable storage. In native OCaml, references are created using the `ref` keyword, dereferencing using the `!` operator, sequencing using the `;` operator, and assignment using the `:=` operator. Conceptually, the difference between the lexical environment for functional programming and imperative programming only differ by their semantics, and that the concept of mutable storage requires an additional "environment" called a *store*. In terms of implementation within this project, the evaluation of expressions containing the impure constructs above utilize a dual-environment method, while the other environments (substitution, dynamic, and lexical) raise errors signifying that their semantic rules do not handle impure programming constructs.

The `Env` module used to create environments for the dynamic and lexical environments from the first part of the environment is re-used for the store environment. With mutable storage, the expressions are effectively evaluated within using an environment which maps variables to their location in memory, and a store which maps a location to a value. The semantic rules for handling each atomic type and construct are slightly different to the lexical environment rules, and have been implemented mechanically as according to the provided rules from the CSCI E-51 course.

5.1 Lexer changes

The lexer file `miniml_lex.mll` did not require significant changes other than adding the keyword `ref` to the `keyword_table` hashtable, and the symbols `!`, `:=`, and `;` to the `sym_table` hashtable. The parsing rules are able to handle

these additional constructs in its current state.

5.2 Parser changes

Again, the changes were almost trivial within the parser file `miniml_parse.mly`. Tokens for each operator were added accordingly, along with their associativity rules - carefully implemented to replicate the behaviour of native OCaml. The grammar rules were also modified to include the newly introduced operators, with the best description of this probably being to look into the code itself.

5.3 Expression and evaluation changes

These impure constructs were implemented into the `expr.mli` and `expr.ml` language expression files by extending the `expr` type definition. Creating a reference was implemented as `Ref of expr`, dereferencing was implemented as `Deref of expr`, assignment was implemented as `Assign of expr * expr`, and sequencing was implemented as `Sequence of expr * expr`. The functions for converting abstract syntax trees and concrete syntax to their string representations were trivially updated to handle these new constructs.

As SUBSTITUTION, DYNAMIC, and LEXICAL environment semantics are not well-defined to handle state change, functions such as `free_vars`, `subst`, `eval_s`, and `eval_env` merely raised an `EvalError` once these constucts were encountered. The evaluation rules for a lexically scoped environment with mutable state are implemented within a function called `eval_e` in the `evaluation.ml` file. As described in the preceding paragraphs, evaluating an impure expression requires two environments, one of them being the store. Hence, the function header for the `eval_e` function would need to be different from those used in previous sections for a functional programming language. In addition to an additional store environment argument, the returned value would need to be an `Env.value` (which could be either a single value, or a closure of a value and an environment), paired with a `Env.env` (the store). This resulted in the following header for the `eval_e` function:

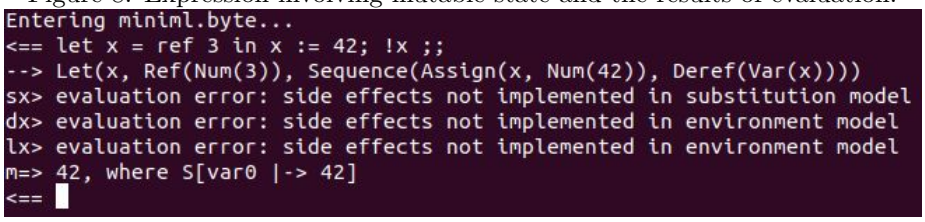
```
let rec eval_e (exp : expr) (env : Env.env) (store : Env.env)
             : Env.value * Env.env =
    ...
```

As a simple demonstration, the following concrete syntax can be entered into MiniML's interpreter, with the result shown in the image below:

```
let x = ref 3 in
x := 42;
!x ;;
```

As seen in Figure 8, the substitution, dynamic environment, and lexical environment semantics raise errors as they are ill-equipped to handle state change. However, the lexically scoped environment handling state change correctly returns the value 42 and the store environment.

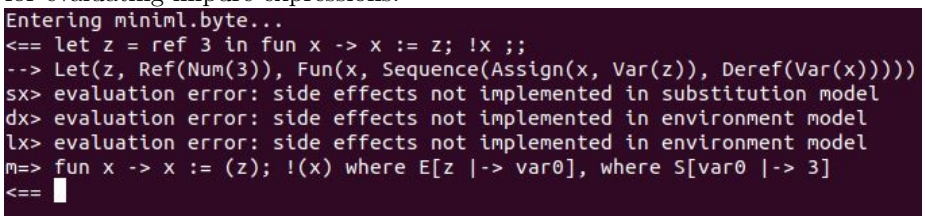
Figure 8: Expression involving mutable state and the results of evaluation.



To illustrate the difference between the environment and the store returned from evaluation of impure expressions, the following example is offered. The REPL is designed to print the results of the store only when an atomic value is returned, keeping consistent with the evaluation rules from the CSCI E-51 course. However, for functions the result is a closure of the function, along with its lexical environment and store. Hence the REPL prints the environment as well in Figure 9, rather than just the value and a store, as seen in Figure 8 when the return value is an atomic type.

```
let z = ref 3 in
fun x -> x := z; !x ;;
```

Figure 9: Example illustrating how the environment and store work in tandem for evaluating impure expressions.



Owing to time pressures in completing this project, I was not able to spend sufficient time to figure out how to implement the chained use of the dereference operator cleanly like native OCaml. For example in MiniML, the expression below needs to implement the dereference operator with white space between the parentheses, which is different to native OCaml where the following is possible: `...!(!a)`, with no whitespaces between the expression symbols. Therefore, this is a topic suggested for future extension to this project.

```
let a = ref 3 in
let b = ref 5 in
let a = ref b in
! ( !a ) ;;
```

6 Conclusion

This document contains a detailed description of the various extensions made to my CSCI E-51 final project, which was to implement a metacircular interpreter for a limited set of constructs based on the OCaml programming language. The first two parts of the assignment to implement SUBSTITUTION and DYNAMIC ENVIRONMENT semantic models are not described in great detail within this document, as the purpose of this document is solely to provide a description of the extensions. Within the time given to complete this project, the following extensions were successfully made:

1. Adding additional atomic types
2. Lexical scoping
3. Syntactic sugar for curried functions
4. References and state change

These extensions were implemented by modifying the lexer file `miniml_lex.mll`, the parser file `miniml_parse.mly`, the expression files `expr.mli` and `expr.ml`, and the evaluator file `evaluation.ml`. As a byproduct of implementing these extensions, a comprehensive suite of unit tests were developed for every function written within the file `tests.ml`, as well as a modification to the REPL file `miniml.ml` to display the evaluation results of different semantic models.

Due to the time constraints of this project, the two main limitations for the extensions implemented are the lack of no-argument functions, and not being able to chain dereference operators cleanly like native OCaml. A detailed overview of these limitations can be found in their relevant subsections within this document. These topics are proposed for future investigation and extensions to this project.