CS51 FINAL PROJECT

STUART M. SHIEBER

Contents

1. Introduction

The culminative final project for CS51 is the implementation of a small subset of an OCaml-like language. Unlike the problem sets, the final project is more open-ended, and we expect you to work more independently, using the skills of design, abstraction, testing, and debugging that you've learned during the course.

1.1. **The MiniML language.** Unlike OCaml and the ML programming language it was derived from, the language you will be implementing includes only a subset of constructs, has only limited support for types (including no user-defined types), and does no type inference (enforcing type constraints only at run-time). On the other hand, the language is "Turing-complete", as expressive as any other programming language in the sense specified by the Church-Turing thesis. (Take CS121 for more on this fundamental and fascinating computational idea.) Because the language is so small, we refer to it as MiniML (pronounced "minimal").

This document introduces the idea of specifying the semantics of a programming language and implementing that specification. The EXERCISES herein are to test your understanding. We recommend that you do the exercises, but you won't be turning them in and we won't be supplying answers. The STAGES provide a sequence of stages to implement MiniML. It's the result of working on these stages that you will be turning in and on which the project grade will be based.

The implementation of this OCaml subset MiniML is in the form of an interpreter for expressions of the language written in OCaml itself, a METACIRCULAR INTERPRETER. Actually, you

Date: April 3, 2018.

will implement a series of interpreters that vary in the semantics they manifest. The first is based on the substitution model; the second a dynamically scoped environment model; and the third, a version of the second implementing one or more extensions of your choosing, with lexical scoping being a simple and highly recommended option.

This project specification is divided into three sections (corresponding to the section numbers marked below):

Substitution model (Section 4): Implementation of a MiniML interpreter using the substitution semantics for the language.

Dynamic scoped environment model (Section 5): Implementation of a MiniML interpreter using the environment model and manifesting dynamic scoping.

Extensions (Section 6): Implementation of one or more extensions to the basic MiniML language of your choosing. Special attention is paid below to an extension to the environment model manifesting lexical scoping (Section 6.2).

1.2. **Grading and collaboration.** As with all the individual problem sets in the course, your project is to be done individually, under the course's standard rules of collaboration. (The sole exception is described in Section 8.) You should not share code with others, nor should you post public questions about your code on Piazza. If you have a clarificatory question about the project assignment, you can post those on Piazza and if appropriate we will answer it publicly so the full class can benefit from the clarification.

The final project will be graded based on correctness of the implementation of the first two stages; design and style of the submitted code; and scope of the project as a whole (including the extensions) as demonstrated by a short paper describing your extensions, which is assessed for both content and presentation.

It may be that you are unable to complete all the code stages of the final project. You should make sure to keep versions at appropriate milestones so that you can always roll back to a working partial project to submit. Using `git` will be especially important for this version tracking if used properly.

Some students or groups might prefer to do a different final project on a topic of their own devising. For students who have been doing exceptionally well in the course to date, this may be possible. See Section 8 for further information.

1.3. **A digression: How is this project different from a problem set?** We frequently get questions about the final project of the following sort: Do I need to implement X? Am I supposed to handle Y? Is it a sufficient extension to do Z? Should I provide tests for W? Is U the right way to handle V? Do I have to discuss P in the writeup?

The final project description doesn't specify answers to many questions of this sort. This is not an oversight; it is a pedagogical choice. In the world of software design and development, there are an infinite number of choices to make, and there are often no right answers, merely tradeoffs. Part of the point of the class is that there are many ways to implement software for a particular purpose, and they are not all equally good. (See Lecture 1, slide 33.) The final project is the place in the course where you are most clearly on your own to deploy the ideas from the class

to make these choices and demonstrate your best understanding of the tradeoffs involved. By implementing X, you may not have time to test Y. By implementing only Z, you may be able to do so with a more elegant or generalizable approach. By adding tests for W, you may not have time to fully discuss P in the writeup. So it goes.

Perhaps the most important of the major tradeoffs is that between spending time to make improvements to the CS51 final project software and writeup and spending time on other non-CS51 efforts. Because choices made in negotiating this tradeoff don't fall solely within the environment of CS51, it is inherently impossible for course staff to give you advice on what to do. You'll have to decide whether your time is better spent, say, systematizing your unit tests for the project, or working on the final paper in your Gen Ed class; further augmenting your implementation of int arithmetic to handle bignums, or studying for the math midterm that the instructor fatuously scheduled during reading period; generating further demonstrations of the mutable array extension you added by implementing a suite of in-place sorting algorithms, or wrangling members of the student organization you find yourself running because the president is AWOL.

With the final project, you are on your own. Not for issues of clarification of this project description, where the course staff stand ready to help on Piazza and in office hours. But on deontic issues, issues of what's better or worse, what you "should" do or mustn't, what is required or forbidden. This is a kind of freedom, and like all freedoms, it is not without consequences, but they are consequences you must inevitably reconcile on your own.

2. Programming language semantics

How do we know that a certain program computes a certain value? When we present OCaml with the expression `3 + 4 * 5`, how do we know that it will calculate `23` as the result? This kind of question is the province of a programming language SEMANTICS. Syntax concerns which expressions of the language are well structured, semantics the meaning of those structured expressions.

SEMANTICS

We'll give the semantics of a language by defining a set of rules that provide a schematic recipe for transforming one expression into another simpler expression. Because the output of each rule is intended to be simpler than the input, the rules are called REDUCTION RULES. The input expression to be transformed is the REDEX, and the output its REDUCTION.

REDUCTION RULES

REDEX

REDUCTION

The idea is probably already familiar to you, as it is implicitly used as early as grade school for providing the semantics of arithmetic expressions. For instance, a sequence of reductions provides a value for the arithmetic expression $3 + 4 \cdot 5$:

$$3 + \underline{4 \cdot 5} \longrightarrow \underline{3 + 20} \longrightarrow 23$$

Here, I've underlined each redex for clarity. The first redex is $4 \cdot 5$, for which we substitute its value 20, implicitly appealing to a reduction rule

$$4 \cdot 5 \longrightarrow 20$$

resulting in an expression which is a redex in its entirety, 3 + 20; that redex further reduces to 23 according to a reduction rule

$$3 + 20 \longrightarrow 23 \quad .$$

You may think these reduction rules seem awfully particular. We'd need an infinite number of them, one for 1 + 1, one for 1 + 2, one for 2 + 1, and so forth. We will shortly generalize them to more schematic rules.

The reduction approach applies to OCaml arithmetic expressions in exactly the same manner, more or less undergoing a mere change of font.

```
3 + 4 * 5 ⟶ 3 + 20 ⟶ 23
```

2.1. **Abstract and Concrete Syntax.** In order to apply the reduction rules properly, we need to have an understanding of the subparts of an expression. The following is an improper application of the reduction rules, because the bit being substituted for, the putative redex, is not a proper subpart of the expression:

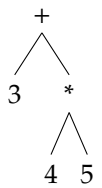
```
3 + 4 * 5 ⟶ 7 * 5 ⟶ 35
```

Expressions are notated as *sequences* of lexical items, the "words" that make up the expressions: `3`, `+`, `4`, `*`, ... We call these elementary lexical units TOKENS. But expressions are really structured hierarchically as *trees* of subexpressions. We distinguish between the ABSTRACT SYNTAX of the language, the tree structures that make up well-formed hierarchical expressions, and the CONCRETE SYNTAX of the language, the token sequences that notate those tree structures. Implicitly, the concrete syntax `3 + 4 * 5` notates the abstract syntax that might be conveyed by the following tree:

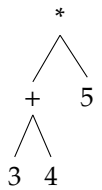
TOKENS

ABSTRACT SYNTAX

CONCRETE SYNTAX



The alternative structure expressed in concrete syntax as `(3 + 4) * 5` would then be



Notice how the abstract syntax has (and needs) no parentheses. Parentheses are a notion of concrete syntax used to explicate the abstract syntax of an expression. From here on, we will use the concrete syntax of OCaml to notate the abstract syntax where convenient, assuming that no confusion should result.

2.2. **Implementing abstract syntax.** For the purpose of writing an interpreter for a language, it is useful to be able to specify the abstract syntax of an expression. An appropriate abstract data type

definition for the abstract syntax of an arithmetic expression language might be given by the following:

```
type expr =
  | Num of int
  | Unop of unop * expr
  | Binop of binop * expr * expr ;;
```

The `unop` and `binop` types enumerate the operators that can be used to form expressions:

```
type unop =
  | Negate ;;
type binop =
  | Plus
  | Minus
  | Times
  | Equals
  | LessThan ;;
```

This allows us to construct abstract syntax trees such as

```
# Binop (Plus, Num 3, Binop (Times, Num 4, Num 5)) ;;
- : expr = Binop (Plus, Num 3, Binop (Times, Num 4, Num 5))
```

(Section 6.3 discusses how a computer system like a compiler or interpreter converts from concrete syntax to abstract syntax, a process known as parsing.)

Exercise 1. *Draw the abstract syntax trees for the following concrete arithmetic expressions.*

(1) `~- 5 * 3`
(2) `3 - 4 - 5`
(3) `1 - 2 / 3 + 4`
(4) `(1 - 2) / (3 + 4)`
(5) `((1 - 2) / (3 + 4))`
(6) `(1 - 2) / 3 + 4`

□

Exercise 2. *Write a function* `exp_to_concrete_string : expr -> string` *that converts an abstract syntax tree of type* `expr` *to a concrete syntax string. The particularities of what concrete syntax you use is not crucial so long as you do something sensible along the lines we've exemplified. (This function will actually be quite helpful in later stages.)* □

2.3. **Schematic reduction rules.** Returning to the reduction semantics for OCaml arithmetic expressions, the reduction rules governing integer arithmetic covering a few binary and unary operators might be

$\underline{m}$ + $\underline{n}$ $\longrightarrow$ $\underline{m+n}$
$\underline{m}$ * $\underline{n}$ $\longrightarrow$ $\underline{m \cdot n}$
$\underline{m}$ - $\underline{n}$ $\longrightarrow$ $\underline{m-n}$

$\underline{m}$ `/` $\underline{n} \longrightarrow \underline{\lfloor m/n \rfloor}$

`~-` $\underline{n} \longrightarrow \underline{-n}$

and so forth. Here, we use the notation $\underline{m}$ for the OCaml numeral token that encodes the number m. (The FLOOR notation $\lfloor x \rfloor$ might be unfamiliar to you. It specifies the integer obtained by rounding a real number down, so $\lfloor 2.71828 \rfloor = 2$.)

FLOOR

These rules may look trivial, but they are not. The first rule specifies that the + operator in OCaml when applied to two numerals has the property of generating the numeral representing their sum. The language being specified might have used the operator ♠ for integer addition, leading to the rule

$\underline{m}$ ♠ $\underline{n} \longrightarrow \underline{m+n}$

making clear the distinction between the object language whose semantics is being defined and the metalanguage being used to define it.

Similarly, the fourth rule specifies that the / operator in OCaml when applied to two numerals specifies the integer portion of their ratio. It could have been otherwise.[1] The language might have used a different operator for integer division,

$\underline{m}$ `//` $\underline{n} \longrightarrow \underline{\lfloor m/n \rfloor}$

(as happens to be used in Python 3 for instance), or could have defined the result differently, say

$\underline{m}$ `/` $\underline{n} \longrightarrow \underline{\lceil m/n \rceil}$.

Nonetheless, there is not *too* much work being done by these rules, and if that were all there were to defining a semantics, there would be little reason to go to the trouble. Things get more interesting, however, when additional constructs such as function application are considered, which we turn to next.

3. Substitution semantics

Consider, for instance, the application of a function to an argument, which would be expressed by a redex of the form `(fun` ⟨*var*⟩ `->` ⟨*body*⟩`)` ⟨*arg*⟩. This allows programs like

```
(fun x -> x + x) (3 * 4)
```

The reduction rule for this construct might be something like

`(fun` x `->` P`)` $Q \longrightarrow P[x \mapsto Q]$

where x stands for an arbitrary variable, and P and Q for arbitrary expressions, and $P[x \mapsto Q]$ is the SUBSTITUTION of Q for x in P.[2] Because of the central place of substitution in providing the semantics of the language, this approach to semantics is called SUBSTITUTION SEMANTICS.

SUBSTITUTION

SUBSTITUTION SEMANTICS

[1] What may be mind-boggling here is the role of the mathematical notation used on the right-hand side of the rule. How is it that we can make use of notations like $\lfloor m/n \rfloor$ in defining the semantics of the / operator? Doesn't appeal to that kind of mathematical notation beg the question? Or at least call for its own semantics? Yes, it does, but since we have to write down the semantics of constructs somehow or other, we use commonly accepted mathematical notation applied in the context of natural language (in the case at hand, English). You may think that this merely postpones the problem of giving OCaml semantics by reducing it to the problem of giving semantics for mathematical notation and English. You would be right, and the problem is further exacerbated when the semantics makes use of mathematical notation that is not so familiar, for instance, the substitution notation to be introduced shortly. But we have to start somewhere.

[2] You may want to refer to the document "On notation" made available at the course website here (for CS51) and here (for CSCI E-51).

Exercise 3. *Give the expressions (in concrete syntax) specified by the following substitutions:*

(1) $(x + x)[x \mapsto 3]$

(2) $(x + x)[y \mapsto 3]$

(3) $(x * x)[x \mapsto 3 + 4]$

□

Let's use this new rule to evaluate the example above:

```
(fun x -> x + x) (3 * 4)
⟶ 3 * 4 + 3 * 4
⟶ 12 + 3 * 4
⟶ 12 + 12
⟶ 24
```

Of course, there is an alternate derivation.

```
(fun x -> x + x) (3 * 4)
⟶ (fun x -> x + x) 12
⟶ 12 + 12
⟶ 24
```

In this case, it doesn't matter which order we apply the rules, but later, it will become important. We can mandate a particular order of reduction by introducing a new concept, the VALUE. An expression is a value if no further reduction rules apply to it. Numerals, for instance, are values. We'll also specify that `fun` expressions are values; there aren't any reduction rules that apply to a `fun` expression, and we won't apply any reduction rules within its body. We can restrict the function application rule to apply only when its argument expression is a value, that is,

VALUE

`(fun` x `->` P`)` $v \longrightarrow P[x \mapsto v]$

(We use the schematic expression v to range only over values.) In that case, the step

`(fun x -> x + x) (3 * 4)` $\longrightarrow$ `3 * 4 + 3 * 4`

doesn't hold because `3 * 4` is not a value. Instead, the second derivation is the only one that applies.

What about OCaml's local naming construct, the `let` $\cdots$ `in` $\cdots$ form? Once we have function application in place, we can give a simple semantics to variable definition by reducing it to function application as per this rewrite rule:

`let` x `=` v `in` $P \longrightarrow$ `(fun` x `->` P`)` v

Exercise 4. *Use the reduction rules developed so far to reduce the following expressions to their values.*

(1)
```
let x = 3 * 4 in
x + x
```

(2)
```
let f = fun x -> x in
f (f 5)
```

□

Exercise 5. *Show that the rule for the* `let` *construct could have been written instead as*

`let` x `=` v `in` $P \longrightarrow P[x \mapsto v]$

From here on, we'll use this rule instead of the previous rule for `let`. □

Let's try a derivation using all these rules.

```
let double = fun x -> 2 * x in double (double 3)
⟶ (fun x -> 2 * x) ((fun x -> 2 * x) 3)
⟶ (fun x -> 2 * x) (2 * 3)
⟶ (fun x -> 2 * x) 6
⟶ 2 * 6
⟶ 12
```

Exercise 6. *Carry out similar derivations for the following expressions:*

(1)
```
let square = fun x -> x * x in
let y = 3 in
square y
```

(2)
```
let id = fun x -> x in
let square = fun x -> x * x in
let y = 3 in
id square y
```

□ 185

You may have noticed in Exercise 6 that some care must be taken when substituting. Consider the following case:

```
let x = 3 in let double = fun x -> x + x in double 4
```

If we're not careful, we'll get a derivation like this:

```
let x = 3 in let double = fun x -> x + x in double 4
⟶ let double = fun x -> 3 + 3 in double 4
⟶ (fun x -> 3 + 3) 4
⟶ 3 + 3
⟶ 6
```

or even worse

```
let x = 3 in let double = fun x -> x + x in double 4
⟶ let double = fun 3 -> 3 + 3 in double 4
⟶ (fun 3 -> 3 + 3) 4
⟶ hunh?
```

It appears we must be very careful in how we define this substitution operation $P[x \mapsto Q]$. In 190
particular, we don't want to replace *every* occurrence of the token x in P, only the *free* occurrences. A variable being introduced in a `fun` should definitely not be replaced, nor should any occurrences of x within the body of a `fun` that also introduces x.

FREE VARIABLES

More precisely, we can define the set of FREE VARIABLES in an expression P, notated $FV(P)$ through the recursive definition in Figure 1. By way of example, the definition says that the free

variables in the expression `fun y -> f (x + y)` are just `f` and `x`, as shown in the following derivation:

$$\begin{aligned} FV(\texttt{fun y -> f (x + y)}) &= FV(\texttt{f (x + y)}) - \{\texttt{y}\} \\ &= FV(\texttt{f}) \cup FV(\texttt{x + y}) - \{\texttt{y}\} \\ &= \{\texttt{f}\} \cup FV(\texttt{x}) \cup FV(\texttt{y}) - \{\texttt{y}\} \\ &= \{\texttt{f}\} \cup \{\texttt{x}\} \cup \{\texttt{y}\} - \{\texttt{y}\} \\ &= \{\texttt{f}, \texttt{x}, \texttt{y}\} - \{\texttt{y}\} \\ &= \{\texttt{f}, \texttt{x}\} \end{aligned}$$

Exercise 7. *Use the definition of FV to derive the set of free variables in the expressions below. Circle all of the free occurrences of the variables.*

(1) `let x = 3 in let y = x in f x y`
(2) `let x = x in let y = x in f x y`
(3) `let x = y in let y = x in f x y`
(4) `let x = fun y -> x in x`

□

Exercise 8. *The definition of FV in Figure 1 is incomplete, in that it doesn't specify the free variables in a* `let rec` *expression. Add appropriate rules for this construct of the language, being careful to note that in an expression like* `let rec x = fun y -> x in x`*, the variable* `x` *is not free. (Compare with Exercise 7(4).)* □

A good start to a definition of the substitution operation is given by the following recursive definition, which replaces only free occurrences of variables:

$$\begin{aligned} \underline{m}[x \mapsto Q] &= \underline{m} \\ x[x \mapsto Q] &= Q \\ y[x \mapsto Q] &= y && \text{where } x \not\equiv y \\ (\texttt{~-}\ P)[x \mapsto Q] &= \texttt{~-}\ P[x \mapsto Q] && \text{and similarly for other unary operators} \\ (P\ \texttt{+}\ R)[x \mapsto Q] &= P[x \mapsto Q]\ \texttt{+}\ R[x \mapsto Q] && \text{and similarly for other binary operators} \\ (\texttt{fun}\ x\ \texttt{->}\ P)[x \mapsto Q] &= \texttt{fun}\ x\ \texttt{->}\ P \\ (\texttt{fun}\ y\ \texttt{->}\ P)[x \mapsto Q] &= \texttt{fun}\ y\ \texttt{->}\ P[x \mapsto Q] && \text{where } x \not\equiv y \end{aligned}$$

Exercise 9. *Use the definition of the substitution operation above to verify your answers to Exercise 3.* □

3.1. **More on capturing free variables.** But there is still a problem in our definition of substitution. Consider the following expression: `let x = y in (fun y -> x)`. Intuitively speaking, this expression seems ill-formed; it defines `x` to be an unbound variable `y`. But using the definitions that we have given so far, we would have the following derivation:

```
let x = y in (fun y -> x)
⟶ (fun x -> (fun y -> x)) y
⟶ (fun y -> x)[x ↦ y]
= fun y -> (x[x ↦ y])          ⇐
= fun y -> y
```

The problem happens in the line marked ⇐. We're sneaking a `y` inside the scope of the variable `y` bound by the `fun`. That's not kosher. We need to change the definition of substitution to make sure that such VARIABLE CAPTURE doesn't occur. The following rules work by replacing the bound variable `y` with a new freshly minted variable, say `z` that doesn't occur elsewhere, renaming all occurrences of `y` accordingly.

VARIABLE CAPTURE

$$\texttt{(fun } x \texttt{ -> } P\texttt{)}[x \mapsto Q] = \texttt{fun } x \texttt{ -> } P$$

$$\texttt{(fun } y \texttt{ -> } P\texttt{)}[x \mapsto Q] = \texttt{fun } y \texttt{ -> } P[x \mapsto Q]$$

where $x \not\equiv y$ and $y \notin FV(Q)$

$$\texttt{(fun } y \texttt{ -> } P\texttt{)}[x \mapsto Q] = \texttt{fun } z \texttt{ -> } P[y \mapsto z][x \mapsto Q]$$

where $x \not\equiv y$ and $y \in FV(Q)$ and z is a fresh variable

Exercise 10. *Carry out the derivation for* `let x = y in (fun y -> x)` *as above but with this updated definition of substitution.* □

Exercise 11. *What should the corresponding rule or rules defining substitution on* `let` $\cdots$ `in` $\cdots$ *expressions be? That is, how should the following rule be completed? You'll want to think about how this construct reduces to function application in determining your answer.*

$$\texttt{(let } y \texttt{ = } P \texttt{ in } R\texttt{)}[x \mapsto Q] = \cdots$$

Try to work out your answer before checking it with the full definition of substitution in Figure 1. □

Exercise 12. *Use the definition of the substitution operation above to determine the results of the following substitutions:*

(1) `(fun x -> x + x)`$[x \mapsto 3]$
(2) `(fun x -> y + x)`$[x \mapsto 3]$
(3) `(let x = y * y in x + x)`$[x \mapsto 3]$
(4) `(let x = y * y in x + x)`$[y \mapsto 3]$

□

3.2. **Substitution semantics of recursion.** You may observe that the rule for evaluating `let` $\cdots$ `in` $\cdots$ expressions

$$\texttt{let } x \texttt{ = } v \texttt{ in } P \longrightarrow P[x \mapsto v]$$

doesn't handle recursion properly. For instance, the Fibonacci example proceeds as follows:

```
let f = fun n -> if n = 0 then 1 else n * f (n-1) in f 2
⟶ (f 2)[f ↦ (fun n -> if n = 0 then 1 else n * f (n-1))]
⟶ (fun n -> if n = 0 then 1 else n * f (n-1)) 2
⟶ if 2 = 0 then 1 else 2 * f (2-1)
```

which eventually leads to an attempt to apply the unbound `f` to its argument `1`.

Occurrences of the definiendum in the body are properly replaced with the definiens, but occurrences in the definiens itself are not. But what should those recursive occurrences of `f` be replaced *by*? It doesn't suffice simply to replace them with the definiens, as that still has a free occurrence of the definiendum. Rather, we'll replace them with their own recursive `let` construction, thereby allowing later occurrences to be handled as well. In particular, the rule for `let rec`, subtly different from the `let` rule, can be as follows:

`let rec` x `=` v `in` $P \longrightarrow P[x \mapsto v[x \mapsto$ `let rec` x `=` v `in` $x]]$

For instance, in the factorial example above, we first replace occurrences of `f` in the definiens with `let rec f = fun n -> if n = 0 then 1 else n * f (n-1) in f`, forming

```
fun n -> if n = 0 then 1
         else n * (let rec f = fun n -> if n = 0
                                        then 1
                                        else n * f (n-1) in f) (n-1)
```

We use that as the expression to replace `f` with in the body `(f 2)`, yielding

```
(fun n -> if n = 0 then 1
          else n * (let rec f = fun n -> if n = 0
                                         then 1
                                         else n * f (n-1) in f) (n-1)) 2
```

Proceeding from there, we derive

```
if 2 = 0 then 1
else 2 * (let rec f = fun n -> if n = 0
                               then 1
                               else n * f (n-1) in f) (2-1))
⟶
if false then 1
else 2 * (let rec f = fun n -> if n = 0
                               then 1
                               else n * f (n-1) in f) (2-1))
⟶
2 * (let rec f = fun n -> if n = 0
                          then 1
                          else n * f (n-1) in f) (2-1))
⟶
2 * (fun n -> if n = 0
              then 1
```

```
              else n * (let rec f = fun n -> if n = 0
                                             then 1
                                             else n * f (n-1) in f) (n-1)) (2-1))
⟶
2 * (fun n -> if n = 0
              then 1
              else n * (let rec f = fun n -> if n = 0
                                             then 1
                                             else n * f (n-1) in f) (n-1)) 1)
⟶
2 * (if 1 = 0
     then 1
     else 1 * (let rec f = fun n -> if n = 0
                                    then 1
                                    else n * f (n-1) in f) (1-1))
⟶
2 * (if false
     then 1
     else 1 * (let rec f = fun n -> if n = 0
                                    then 1
                                    else n * f (n-1) in f) (1-1))
```

Exercise 13. *Thanklessly continue this derivation until it converges on the final result for the factorial of 2, viz., 2. Then thank your lucky stars that we have computers to do this kind of rote repetitive task for us.* □ 235

4. Implementing a substitution semantics for MiniML

You'll start your implementation with a substitution semantics for MiniML. The abstract syntax of the language is given by the following type definition:

```
type expr =
  | Var of varid                        (* variables *)
  | Num of int                          (* integers *)
  | Bool of bool                        (* booleans *)
  | Unop of unop * expr                 (* unary operators *)
  | Binop of binop * expr * expr        (* binary operators *)
  | Conditional of expr * expr * expr   (* if then else *)
  | Fun of varid * expr                 (* function definitions *)
  | Let of varid * expr * expr          (* local naming *)
  | Letrec of varid * expr * expr       (* recursive local naming *)
  | Raise                               (* exceptions *)
  | Unassigned                          (* (temporarily) unassigned *)
```

```
  | App of expr * expr                    (* function applications *)
 and varid = string ;;
```

(The `unop` and `binop` enumerated types are as above.) These type definitions can be found in the partially implemented `Expr` module in the files `expr.ml` and `expr.mli`. You'll notice that the module signature requires additional functionality that hasn't been implemented, including functions to find the free variables in an expression, to generate a fresh variable name, and to substitute expressions for free variables, as well as to generate various string representations of expressions.

To get things started, we also provide a parser for the MiniML language, which takes a string in a concrete syntax and returns a value of this type `expr`; you may want to extend the parser in a later part of the project (Section 6.3). The compiled parser and a read-eval-print loop for the language are available in the following files:

evaluation.ml: The future home of anything needed to evaluate expressions to values. Currently, it provides a trivial "evaluator" `eval_t` that merely returns the expression unchanged.

miniml.ml: Runs a read-eval-print loop for MiniML, using the Evaluation module that you will write.

miniml_lex.mll: A lexical analyzer for MiniML. (You should never need to look at this unless you want to extend the parser.)

miniml_parse.mly: A parser for MiniML. (Ditto.)

What's left to implement is the `Evaluation` module in `evaluation.ml`.

Start by familiarizing yourself with the code. You should be able to compile `miniml.ml` and get the following behavior.[3]

```
# ocamlbuild miniml.byte
Finished, 13 targets (12 cached) in 00:00:00.
# ./miniml.byte
Entering miniml.byte...
<== 3 ;;
Fatal error: exception Failure("exp_to_abstract_string not implemented")
```

Stage 1. *Implement the function* `exp_to_abstract_string : expr -> string` *to convert abstract syntax trees to strings representing their structure and test it thoroughly. If you did Exercise 2, the experience may be helpful here, and you'll want to also implement* `exp_to_concrete_string : expr -> string` *for use in later stages as well. The particularities of what concrete syntax you use to depict the abstract syntax is not crucial – we won't be checking it – so long as you do something sensible along the lines we've exemplified.*

[3] In building the project, you may find that you get a warning of the form:

```
+ /Users/uesrname/.opam/4.06.0/bin/ocamlyacc miniml_parse.mly
185 shift/reduce conflicts, 18 reduce/reduce conflicts.
```

You can safely ignore this message from the parser generator, which is reporting on some ambiguities in the MiniML grammar that it has resolved automatically.

After this (and each) stage, it would be a good idea to commit the changes and push to your remote repository as a checkpoint and backup. □

Once you write the function `exp_to_abstract_string`, you should have a functioning read-eval-print loop, except that the evaluation part doesn't do anything. (The REPL calls the trivial evaluator `eval_t`, which essentially just returns the expression unchanged.) Consequently, it just prints out the abstract syntax tree of the input concrete syntax:

```
# ./miniml.byte
Entering miniml.byte...
<== 3 ;;
==> Num(3)
<== 3 4 ;;
==> App(Num(3), Num(4))
<== (((3) ;;
xx> parse error
<== let f = fun x -> x in f f 3 ;;
==> Let(f, Fun(x, Var(x)), App(App(Var(f), Var(f)), Num(3)))
<== let rec f = fun x -> if x = 0 then 1 else x * f (x - 1) in f 4 ;;
==> Letrec(f, Fun(x, Conditional(Binop(Equals, Var(x), Num(0)), Num(1),
    Binop(Times, Var(x), App(Var(f), Binop(Minus, Var(x), Num(1))))),
    App(Var(f), Num(4)))
<== Goodbye.
```

Exercise 14. *Familiarize yourself with how this "almost" REPL works. How does `eval_t` get called? What does `eval_t` do and why? What's the point of the `Env.Val` in the definition? Why does `eval_t` take an argument `_env : Env.env`, which it just ignores? (These last two questions are answered a few paragraphs below. Feel free to read ahead.)* □

To actually get evaluation going, you'll need to implement a substitution semantics, which requires completing the functions in the `Expr` module.

Stage 2. *Start by writing the function `free_vars` in `expr.ml`, which takes an expression (`expr`) and returns a representation of the free variables in the expression, according to the definition in Figure 1. Test this function completely.* □

Stage 3. *Next, write the function `subst` that implements substitution as defined in Figure 1. In some cases, you'll need the ability to define new fresh variables in the process of performing substitutions. Something like the `gensym` function that you wrote in lab might be useful for that. Once you've written `subst` make sure to test it completely.* □

You're actually quite close to having your first working interpreter for MiniML. All that is left is writing a function `eval_s` (the 's' is for *substitution semantics*) that evaluates an expression using the substitution semantics rules. (Those rules were described informally in lecture 7. The lecture slides may be helpful to review.) The `eval_s` function walks an abstract syntax tree of type `expr`,

evaluating subparts recursively where necessary and performing substitutions when appropriate. The recursive traversal bottoms out when you get to primitive values like numbers or booleans or in applying primitive functions like the unary or binary operators to values. It is at this point that the evaluator can see if the operators are being applied to values of the right type, integers for the arithmetic operators, for instance, or integers or booleans for the comparison operators.

For consistency with the environment semantics that you will implement later as the function `eval_d`, both `eval_t` and `eval_s` take a second argument, an environment, even though neither evaluator needs an environment. Thus your implementation of `eval_s` can just ignore the environment.

We'd also like the various evaluation functions `eval_t`, `eval_s`, `eval_d`, and (if implemented) `eval_l` to all have the same return type as well. Looking ahead, the lexically-scoped environment semantics implemented in `eval_l` must allow for the result of evaluation to go beyond the simple expression values we've used so far. In particular, we'll want to add closures as a new sort of value, as described in Section 6.2. We've provided a variant type `Env.value` that allows for both the simple expression values of the sort that `eval_s` and `eval_d` generate and for closures, which only the environment-based evaluators generate. For consistency, then, you should make sure that `eval_s`, as well as the later evaluation functions, are of type `Expr.expr -> Env.env -> Env.value`. This will ensure that your code is consistent with our unit tests as well. You'll note that the `eval_t` evaluator that we provide already does this. In order to be type-consistent, it takes an extra `env` argument that it doesn't need or use, and converts its `expr` argument to the `value` type by adding the `Env.Val` value constructor for that type. (This may help with Exercise 14.)

Stage 4. *Implement the* `eval_s : Expr.expr -> Env.env -> Env.value` *function in* `evaluation.ml`*. (You can hold off on completing the implementation of the Env module for the time being. That comes into play in later sections.) We recommend that you implement it in stages, from the simplest bits of the language to the most complex. You'll want to test each stage thoroughly using unit tests as you complete it. Keep these unit tests around so that you can easily unit test the later versions of the evaluator that you'll develop in future sections.* □

Using the substitution semantics, you should be able to handle evaluation of all of the MiniML language. If you want to postpone handling of some parts while implementing the evaluator, you can always just raise the `EvalError` exception, which is intended just for this kind of thing, when a runtime error occurs. Another place `EvalError` will be useful is when a runtime type error occurs, for instance, for the expressions `3 + true` or `3 4` or `let x = true in y`.

Now that you have implemented a function to evaluate expressions, you can make the REPL loop worthy of its name. Notice at the bottom of `evaluation.ml` the definition of `evaluate`, which is the function that the REPL loop in `miniml.ml` calls. Replace the definition with the one calling `eval_s` and the REPL loop will evaluate the read expression before printing the result. It's more pleasant to read the output expression in concrete rather than abstract syntax, so you can replace the `exp_to_abstract_string` call with a call to `exp_to_concrete_string`. You should end up with behavior like this:

```
# miniml_soln.byte
```

```
Entering miniml_soln.byte...
<== 3 ;;
==> 3
<== 3 + 4 ;;
==> 7
<== 3 4 ;;
xx> evaluation error: (3 4) bad redex
<== (((3) ;;
xx> parse error
<== let f = fun x -> x in f f 3 ;;
==> 3
<== let rec f = fun x -> if x = 0 then 1 else x * f (x - 1) in f 4 ;;
xx> evaluation error: not yet implemented: let rec
<== Goodbye.
```

Some things to note about this example:

- The parser that we provide will raise an exception `Parsing.Parse_error` if the input doesn't parse as well-formed MiniML. The REPL handles the exception by printing an appropriate error message.
- The evaluator can raise an exception `Evaluation.EvalError` at runtime if a (well-formed) MiniML expression runs into problems when being evaluated.
- You might also raise `Evaluation.EvalError` for parts of the evaluator that you haven't (yet) implemented, like the tricky `let rec` construction in the example above.

Stage 5. *After you've changed* `evaluate` *to call* `eval_s`*, you'll have a complete working implementation of MiniML. As usual, you should save a snapshot of this using a git commit and push so that if you have trouble down the line you can always roll back to this version to submit it.* □

5. Implementing an environment semantics for MiniML

The substitution semantics is sufficient for all of MiniML because it is a pure functional programming language. But binding constructs like `let` and `let rec` are awkward to implement, and extending the language to handle references, mutability, and imperative programming is impossible. For that, you'll extend the language semantics to make use of an ENVIRONMENT that stores a mapping from variables to their values. You will want to be able to replace the values of variables dynamically – you'll see why shortly – so that these variable values need to be mutable. We've provided a type signature for environments. It stipulates types for environments and values, and functions to create an empty environment (which we've already implemented for you), to extend an environment with a new BINDING, that is, a mapping of a variable to its (mutable) value, and to look up the value associated with a variable.

ENVIRONMENT

BINDING

Stage 6. *Implement the various functions involved in the* `Env` *module and test them thoroughly.* □

How will this be used? For atomic literals like numerals and truth values, they evaluate to themselves as usual, independently of the environment. But to evaluate a variable in an environment, we look up the value that the environment assigns to it and return that value.

A slightly more complex case involves function application, as in this example:

```
(fun x -> x + x) 5
```

The abstract syntax for this expression is an application of one expression to another.

To evaluate an application $P\ Q$ in an environment ρ,

(1) Evaluate P in ρ to a value v_P, which should be a function `fun x -> B`. If v_P is not a function, raise an evaluation error.
(2) Evaluate Q in the environment ρ to a value v_Q.
(3) Evaluate B in the environment obtained by extending ρ with a binding of `x` to v_Q.

In the example: (1) `fun x -> x + x` is already a function, so evaluates to itself. (2) The argument `5` also evaluates to itself. (3) The body `x + x` is evaluated in an environment that maps `x` to `5`.

For `let` expressions, a similar evaluation process is used. Consider

```
let x = 3 * 4 in x + 1 ;;
```

The abstract syntax for this `let` expression has a variable name, a definition expression, and a body. To evaluate this expression in, say, the empty environment, we first evaluate (recursively) the definition part in the same empty environment, presumably getting the value `12` back. We then extend the environment to associate that value with the variable `x` to form a new environment, and then evaluate the body `x + 1` in the new environment. In turn, evaluating `x + 1` involves recursively evaluating `x` and `1` in the new environment. The latter is straightforward. The former involves just looking up the variable in the environment, retrieving the previously stored value `12`. The sum can then be computed and returned as the value of the entire `let` expression.

For recursion, consider this expression, which makes use of an (uninteresting) recursive function:

```
let rec f = fun x -> if x = 0 then x else f (x - 1) in f 2 ;;
```

Again, the `let rec` expression has three parts: a variable name, a definition expression, and a body. To evaluate it, we ought to first evaluate the definition part, but using what environment? If we use the incoming (empty) environment, then what will we use for a value of `f` when we reach it? We should use the value of the definition, but we don't have it yet. In the interim, we'll store a special value, `Unassigned`, which you'll have noticed in the `expr` type but which is never generated by the parser. We evaluate the definition in this extended environment, hopefully generating a value. (The definition part better not ever evaluate the variable name though, as it is unassigned; doing so should raise an `EvalError`. An example of this problem might be `let rec x = x in x`.) The value returned for the definition can then *replace* the value for the variable name (thus the need for a mutable environment) and that environment passed on to the body for evaluation.

In the example above, we augment the empty environment with a binding for `f` to `Unassigned` and evaluate `fun x -> if x = 0 then x else f (x - 1)` in that environment. Since this is a function, it is already a value, and the environment can be updated to have `f` have

this function as a value. Finally, the body `f 2` is evaluated in this environment. The body, an application, evaluates `f` by looking it up in this environment yielding `fun x -> if x = 0 then x else f (x - 1)` and evaluates `2` to itself, then evaluates the body of the function in the prevailing environment (in which `f` has its value) augmented with a binding of `x` to `2`.

Stage 7. *Implement another evaluation function* `eval_d : Expr.expr -> Env.env -> Env.value` *(the '*`d`*' is for* dynamically scoped environment semantics*), which works along the lines just discussed. Make sure to test it on a range of tests exercising all the parts of the language.* □

6. Extending the language

In this final part of the project, you will extend MiniML in one or more ways of your choosing.

6.1. Extension ideas.

Here are a few ideas for extending the language, very roughly in order from least to most ambitious. Especially difficult extensions are marked with ❢ symbols.

(1) Add additional atomic types (floats, strings, unit, etc.) and corresponding literals and operators.
(2) Modify the environment semantics to manifest lexical scope instead of dynamic scope (Section 6.2).
(3) Augment the syntax by allowing for one or more bits of syntactic sugar, such as the curried function definition notation seen in `let f x y z = x + y * z in f 2 3 4`.
(4) Add lists to the language.
(5) Add records to the language.
(6) Add references to the language, by adding operators `ref`, `!`, and `:=`.
(7) Add laziness to the language (by adding refs and syntactic sugar for the `lazy` keyword). If you've also added lists, you'll be able to build infinite streams.
(8) Add better support for exceptions, for instance, multiple different exception types, exceptions with arguments, exception handling with `try...with...`.
(9) ❢ Add simple compile-time type checking to the language. For this extension, the language would be extended so that *every* introduction of a bound variable (in a `let`, `let rec`, or `fun` construct) is accompanied by its (monomorphic) type. The abstract syntax would need to be extended to store those types, and you would write a function to walk the tree to verify that every expression in the program is well typed. This is a quite ambitious project.
(10) ❢❢ Add type inference to the language, so that (as in OCaml) types are inferred even when not given explicitly. This is *extremely ambitious*, not for the faint of heart. Do not attempt to do this.

Most of the extensions (in fact, all except for (2)) require extensions to the concrete syntax of the language. We provide information about extending the concrete syntax in Section 6.3. Many other extensions are possible. Don't feel beholden to this list. Be creative!

In the process of extending the language, you may find the need to expand the definition of what an expression is as codified in the file `expr.mli`. Other modifications may be necessary as well. That is, of course, expected, but you should make sure that you do so in a manner

compatible with the existing codebase so that unit tests based on the provided definitions continue to function. The ability to submit your code for testing should help with this process.

Most importantly: It is better to do a great job (clean, elegant design; beautiful style; well thought-out implementation; evocative demonstrations of the extended language; literate writeup) on a smaller extension, than a mediocre job on an ambitious extension. That is, the scope aspect of the project will be weighted substantially less than the design and style aspects. Caveat scriptor.

6.2. **A lexically scoped environment semantics.** One possible extension is to implement a lexically scoped environment semantics, perhaps with some further extensions. Consider the following OCaml expression:

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

Exercise 15. *What should this expression evaluate to? Test it in the OCaml interpreter. Try this expression using your* `eval_s` *and* `eval_d` *evaluators. Which ones accord with OCaml's evaluation?* □

The `eval_d` evaluator that you've implemented so far is DYNAMICALLY SCOPED. The values of variables are governed by the dynamic ordering in which they are evaluated. But OCaml is LEXICALLY SCOPED. The values of variables are governed by the lexical structure of the program. In the case above, when the function `f` is applied to `3`, the most recent assignment to `x` is of the value `2`, but the assignment to the `x` that lexically outscopes `f` is of the value `1`. Thus a dynamically scoped language calculates the body of `f`, `x + y`, as `2 + 3` (that is, `5`) but a lexically scoped language calculates the value as `1 + 3` (that is, `4`).

The substitution semantics manifests lexical scope, as it should, but the dynamic semantics does not. To fix the dynamic semantics, we need to handle function values differently. When a function value is computed (say the value of `f`, `fun y -> x + y`), we need to keep track of the lexical environment in which the function occurred so that when the function is eventually applied to an argument, we can evaluate the application in the lexical environment – the environment when the function was *defined* – rather than the dynamic environment – the environment when the function was *called*.

The technique to enable this is to package up the function being defined with a snapshot of the environment at the time of its definition into a data structure called a CLOSURE. There is already provision for closures in the `env` module. You'll notice that the `value` type has two constructors, one for normal values (like numbers, booleans, and the like) and one for closures. The `Closure` constructor just puts together a function with its lexical environment.

Stage 8 (if you decide to do a lexically scoped evaluator in service of your extension)**.** *Make a copy of your* `eval_d` *evaluation function and call it* `eval_l` *(the '*`l`*' for* lexically scoped environment semantics*). Modify the code so that the evaluation of a function returns a closure containing the function itself and the current environment. Modify the function application part so that it evaluates the body of the function in the lexical environment from the corresponding closure (appropriately updated). As usual, test it*

thoroughly. If you've carefully accumulated good unit tests for the previous evaluators, you should be able to fully test this new one with just a single function call.

The copy-paste recommendation for building `eval_l` *from* `eval_d` *makes for simplicity in the process, but will undoubtedly leave you with redundant code. Once you've got this all working, you may want to think about merging the two implementations so that they share as much code as possible. Various of the abstraction techniques you've learned in the course could be useful here.* □

6.3. **The MiniML parser.** We provided you with a MiniML parser that converts the concrete syntax of MiniML to an abstract syntax representation using the `expr` type. But to extend the implemented language, you'll typically need to extend the parser. Feel free to do so, but make sure that you extend the language by adding new constructs to the `expr` type, without changing the ones that are already given. For instance, if you want to add support for multiple exceptions, you'll want to leave the `Raise` construct as is (so we can test it with our unit tests) and add your own new construct, say `RaiseExn` for the extension.

The parser we provided was implemented using `ocamllex` and `ocamlyacc`, programs designed to build lexical analyzers and parser for programming languages. Documentation for them can be found at http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html, and tutorial material is archived at https://web.archive.org/web/20150921125456/http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial and https://web.archive.org/web/20150907071659/http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial/.

In summary, `ocamllex` takes a specification of the tokens of a programming language in a file, say, `miniml_lex.mll`. The `ocamlbuild` system knows how to use `ocamllex` to turn such files into OCaml code for a lexical analyzer in the file `miniml_lex.ml`. Similarly, an `ocamlyacc` specification of a parser in a file `miniml_parse.mly` will be transformed by `ocamlyacc` (automatically with `ocamlbuild`) to a parser in `miniml_parse.ml`. By modifying `miniml_lex.mll` and `miniml_parse.mly`, you can modify the concrete syntax of the MiniML language, which may be useful for many of the extensions you might be interested in.

7. Submitting the project

Stage 9. *Write up your extensions in a short paper describing and demonstrating any extensions and how you implemented them. Use Markdown or LaTeX format, and name the file* `writeup.md` *or* `writeup.tex`*. You'll submit both the source file and a rendered PDF file.* □

In addition to submitting the code implementing MiniML to the course grading server through the normal process, you should submit the `writeup.md` or `writeup.tex` file and the rendered PDF file `writeup.pdf` as well.

8. Alternative final projects

Students who have been doing exceptionally well in the course to date can petition to do alternative final projects of their own devising, under the following stipulations:

(1) Alternative final projects can be undertaken individually or in groups of up to four.

(2) The implementation language for the project must be OCaml.
(3) You will want to talk to course staff about your ideas early to get initial feedback.
(4) You will need to submit a proposal for the project by April 13, 2018. The proposal should describe what the project goals are, how you will go about implementing the project, and how the work will be distributed among the members of the group (if applicable).
(5) You will receive notification around April 16, 2018 as to whether your request has been approved. Approval will be based on performance in the course to date and the appropriateness of the project.
(6) You will submit a progress report by April 23, 2018, including a statement of progress, any code developed to date, and any changes to the expected scope of the project.
(7) You will submit the project results, including all code, a demonstration of the project system in action, and a paper describing the project and any results, by May 2, 2018.
(8) You will be scheduled to perform a presentation and demonstration of your project for course staff during reading period.
(9) The group as a whole may drop out of the process at any time. Individual members of the group would then submit instead the standard final project described here.

$$
\begin{aligned}
FV(\underline{m}) &= \varnothing && \text{(integers and other literals)}\\
FV(x) &= \{x\} && \text{(variables)}\\
FV(\texttt{\sim-}\ Q) &= FV(Q) && \text{(and similarly for other unary operators)}\\
FV(P\ \texttt{+}\ Q) &= FV(P) \cup FV(Q) && \text{(and similarly for other binary operators)}\\
FV(P\ Q) &= FV(P) \cup FV(Q) && \text{(applications)}\\
FV(\texttt{fun}\ x\ \texttt{->}\ P) &= FV(P) - \{x\} && \text{(functions)}\\
FV(\texttt{let}\ x\ \texttt{=}\ P\ \texttt{in}\ Q) &= (FV(Q) - \{x\}) \cup FV(P) && \text{(binding)}
\end{aligned}
$$

$$
\begin{aligned}
\underline{m}[x \mapsto P] &= \underline{m}\\
x[x \mapsto P] &= P\\
y[x \mapsto P] &= y \qquad \text{where } x \not\equiv y\\
(\texttt{\sim-}\ Q)[x \mapsto P] &= \texttt{\sim-}\ Q[x \mapsto P]\\
&\qquad \text{and similarly for other unary operators}\\
(Q\ \texttt{+}\ R)[x \mapsto P] &= Q[x \mapsto P]\ \texttt{+}\ R[x \mapsto P]\\
&\qquad \text{and similarly for other binary operators}\\
(\texttt{fun}\ x\ \texttt{->}\ Q)[x \mapsto P] &= \texttt{fun}\ x\ \texttt{->}\ Q\\
(\texttt{fun}\ y\ \texttt{->}\ Q)[x \mapsto P] &= \texttt{fun}\ y\ \texttt{->}\ Q[x \mapsto P]\\
&\qquad \text{where } x \not\equiv y \text{ and } y \notin FV(P)\\
(\texttt{fun}\ y\ \texttt{->}\ Q)[x \mapsto P] &= \texttt{fun}\ z\ \texttt{->}\ Q[y \mapsto z][x \mapsto P]\\
&\qquad \text{where } x \not\equiv y \text{ and } y \in FV(P) \text{ and } z \text{ is a fresh variable}\\
(\texttt{let}\ x\ \texttt{=}\ Q\ \texttt{in}\ R)[x \mapsto P] &= \texttt{let}\ x\ \texttt{=}\ Q[x \mapsto P]\ \texttt{in}\ R\\
(\texttt{let}\ y\ \texttt{=}\ Q\ \texttt{in}\ R)[x \mapsto P] &= \texttt{let}\ y\ \texttt{=}\ Q[x \mapsto P]\ \texttt{in}\ R[x \mapsto P]\\
&\qquad \text{where } x \not\equiv y \text{ and } y \notin FV(P)\\
(\texttt{let}\ y\ \texttt{=}\ Q\ \texttt{in}\ R)[x \mapsto P] &= \texttt{let}\ z\ \texttt{=}\ Q[x \mapsto P]\ \texttt{in}\ R[y \mapsto z][x \mapsto P]\\
&\qquad \text{where } x \not\equiv y \text{ and } y \in FV(P) \text{ and } z \text{ is a fresh variable}
\end{aligned}
$$

FIGURE 1. Definitions of free variables and substitution, for a large portion of MiniML. You'll need to extend the definitions for any additional constructs.

Index