21
Final project: Implementing MiniML

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.

21.1 Overview

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. Because the language is so small, we refer to it as MiniML (pronounced "minimal").

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 will implement a series of interpreters that vary in the semantics they manifest. The first is based on the substitution model (Chapter 13); the second a dynamically scoped environment model (Chapter 19); 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 chapter builds on the idea of specifying the semantics of a programming language and implementing that specification begun in Chapters 13 and 19. 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 nine stages to implement the MiniML interpreter.

It's the result of working on these stages that you will be turning in and on which the project grade will be based.

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

Substitution model (Section 21.2) Implementation of a MiniML interpreter using the substitution semantics for the language.

Dynamic scoped environment model (Section 21.3) Implementation of a MiniML interpreter using the environment model and manifesting dynamic scoping.

Extensions (Section 21.4) 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 21.4.2).

21.1.1 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 21.6.) You should not share code with others, nor should you post public questions about your code on Piazza. If you have clarificatory questions about the project assignment, you can post those on Piazza and if appropriate we will answer them 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 21.6 for further information.

21.1.2 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 course is that there are many ways to implement software for a particular purpose, and they are not all equally good. (See Section 1.2.) The final project is the place in the course where you are most clearly on your own to deploy the ideas from the course 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 course; 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.

21.2 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 unop =
  | Negate ;;

type binop =
  | Plus
  | Minus
  | Times
  | Equals
  | LessThan ;;

type varid = string ;;

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 def'ns *)
  | Let of varid * expr * expr        (* local naming *)
  | Letrec of varid * expr * expr     (* rec. local naming *)
  | Raise                             (* exceptions *)
  | Unassigned                        (* (temp) unassigned *)
  | App of expr * expr ;;             (* function app'ns *)
```

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.

Exercise 233 *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.)* □

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 21.4.3).[1] 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 complete.

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.[2]

```
# ocamlbuild -use-ocamlfind 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 233, 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.*

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.)

[1] The parser that we provide makes use of the OCaml package `menhir`, which is a parser generator for OCaml. You should have installed it as per the setup instructions provided at the start of the course by running the following `opam` command:

```
% opam install -y menhir
```

The `menhir` parser generator will be discussed further in Section 21.4.3.

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

```
+ menhir -ocamlc 'ocamlfind ocamlc -thread
    -strict-sequence -package graphics -package
    CS51Utils -w A-4-33-40-41-42-43-34-44' -infer
    miniml_parse.mly
Warning:  15 states have shift/reduce conflicts.
Warning:  one state has reduce/reduce conflicts.
Warning:  198 shift/reduce conflicts were arbitrarily
    resolved.
Warning:  18 reduce/reduce conflicts were arbitrarily
    resolved.
```

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.

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 234 *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 13.3. Test this function completely.* □

Stage 3 *Next, write the function* `subst` *that implements substitution as defined in Figure 13.4. In some cases, you'll need the ability to define new fresh variables in the process of performing substitutions. You'll see we call for a function* `new_varname` *to play that role. Looking at 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 are, conveniently, described

in detail in Chapter 13, and summarized in Figure 13.5.) 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 it gets 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, for the lexical environment semantics, we'll want to add closures as a new sort of value, as described in Section 21.4.2. 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 lexical-scoped evaluator needs to 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 it converts its `expr` argument to the `value` type by adding the `Env.Val` value constructor for that type. (This may help with Exercise 234.)

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 MiniML 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.* □

21.3 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, as described in Chapter 19. 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.

The implementation of environments for the purpose of this project follows that described in Section 19.5. We make use of an environment that allows the values to be mutable:

```
type env = (varid * value ref) list
```

This will be helpful in the implementation of recursion.

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

How will these environments be used? Atomic literals – like numerals and truth values – 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. Recall the environment semantics rule for appli-

cations from Figure 19.1:

$$E \vdash P\ \ Q \Downarrow$$
$$\left|\begin{array}{l} E \vdash P \Downarrow \texttt{fun}\ \ x\ \texttt{->}\ B \\ E \vdash Q \Downarrow v_Q \\ E\{x \mapsto v_Q\} \vdash B \Downarrow v_B \end{array}\right. \qquad (R_{app})$$
$$\Downarrow v_B$$

According to this rule, to evaluate an application $P\ Q$ in an environment E,

1. Evaluate P in E to a value v_P, which should be a function of the form `fun x -> B`. If v_P is not a function, raise an evaluation error.

2. Evaluate Q in the environment E to a value v_Q.

3. Evaluate B in the environment obtained by extending E with a binding of x to v_Q.

The formal semantics rule translates to what is essentially pseudocode for the interpreter.

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 thus evaluated in an environment that maps `x` to `5`.

For `let` expressions, a similar evaluation process is used. Recall the semantics rule:

$$E \vdash \texttt{let}\ \ x\ \texttt{=}\ D\ \texttt{in}\ \ B \Downarrow$$
$$\left|\begin{array}{l} E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B \end{array}\right. \qquad (R_{let})$$
$$\Downarrow v_B$$

We'll apply this rule in evaluating an expression like

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

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.

Don't be surprised that this dynamically scoped evaluator exhibits all of the divergences from the substitution-based evaluator that were discussed in Section 19.2.1. For instance, the evaluator will return different values for certain expressions; it will allow `let`-bound variables to be used recursively; and it will fail on simple curried functions. That's fine. Indeed, it's a sign you've implemented the dynamic scope regime correctly. But it does motivate implementation of a lexical-scoped version of the evaluator described below.

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.* □

21.4 Extending the language

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

21.4.1 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 21.4.2).
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 `:=`. Since the environment is already mutable, you can even implement this extension without implementing stores and modifying the type of the `eval` function, though you may want to anyway.
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 21.4.3. 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. In particular, if you have to make changes to `mli` files, you'll want to do so in a way that extends the signature, rather than restricting it.

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.

21.4.2 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, reproduced from Section 19.2.2:

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

Exercise 235 *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. (See Section 19.2.2 for further discussion.) 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 that 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 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 packages 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.*

Do not just modify `eval_d` *to exhibit lexical scope, as this will cause our unit tests for* `eval_d` *(which assume that it is dynamically scoped) to fail. That's why we ask you to define the lexically scoped evaluator*

as `eval_l`*. 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.* □

Implementing recursion in the lexically-scoped evaluator By far the trickiest bit of implementing lexical scope is the treatment of recursion, so we address it separately. 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 ;;
```

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? Ideally, we should use the value of the definition, but we don't have it yet.

Following the approach described in Section 19.6.1, in the interim, we'll extend the environment with a special value, `Unassigned`, as the value of the variable being recursively defined. You may have noticed this special value in the `expr` type; uniquely, it 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 run-time error 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 environments to map variables to *mutable* values) and the environment can then be used in evaluating the body.

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, so evaluates to itself. (Notice how we never had to evaluate `f` in generating this value.)

Now the environment can be updated to have `f` have this function as a value – not extended (using the `extend` function) but *actually modified* by replacing the value stored in the `value ref` associated with `f` in the environment. 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`.

In summary, a `let rec` expression like `let rec x = D in B` is evaluated via the following five-step process:

1. Extend the incoming environment with a binding of `x` to `Unassigned`; call this extended environment `env_x`.
2. Evaluate the definition subexpression `D` in that environment to get a value `v_D`.
3. Mutate `env_x` so that `x` now maps to `v_D`.
4. Evaluate the body subexpression `B` to get a value `v_B`.
5. Return `v_B`.

21.4.3 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 `menhir`, 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, http://cambium.inria.fr/~fpottier/menhir/manual.html, and tutorial material is available at https://ohama.github.io/ocaml/ocamllex-tutorial/ and https://dev.realworldocaml.org/parsing-with-ocamllex-and-menhir.html.

In summary, `ocamllex` takes a specification of the tokens of a programming language in a file, in our case `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, a `menhir` specification of a parser in a file `miniml_parse.mly` will be transformed by `menhir` (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.