Chapter 9

MiniScheme

Consider the following Scheme expression

```
(let (($x_1$ $e_1$)
      ($x_2$ $e_2$)
       ⋮
      ($x_n$ $e_n$))
  $e$)
```

What is the scope of x_1? What is the scope of x_2?

Based on our experience with Scheme and the English descriptions we've used to reason about how to evaluate expressions, we might conclude that the variables x_i scope *only* over the body of the `let`, i.e., that the x_i's scope only over e. However, there are other reasonable possibilities. For example, each x_i could scope over all expressions $e_{i+1}, \ldots, e_n, e$, i.e., each x_i could scope over *every subsequent* expression in the list of bindings *as well as* the body of the `let`. In fact, this is what `let*` does (see SICP, Exercise 4.7). Another possible option is a *recursive* let, in which each x_i scopes over *all* sub-expressions. This would be useful if we wanted to define mutually-recursive function.

When defining a language, we need to be completely unambiguous about what different language features really mean—that is, we need to have a clear and precise *semantics*! If the semantics aren't clear, then not only is it hard for programmers to figure out what a program should really be doing, but we could end up with an implementation that does something other than what was intended by the author of the language—or we could end up with different implementations that disagree about the meaning of a program!

Operational semantics gives us a tool for precisely defining a semantics. When we use operational semantics, our primary goal is not to be understood, but to not be *misunderstood.* Of course we want our semantics to be understandable, but we *definitely* want it to be be impossible for reasonable people to disagree about what the operational semantics says. That is, if you and I disagree about the meaning of a program, we should be able to sit down, use the operational semantics to determine what a program really does, and both ambiguously reach the same conclusion. An

$$
\begin{array}{rcll}
e, f & ::= & \texttt{true} & \\
 & | & \texttt{false} & \\
 & | & n & \text{integer} \\
 & | & op & \text{primitive operator} \\
 & | & x & \text{variable} \\
 & | & \texttt{(if } e \ e \ e\texttt{)} & \\
 & | & \texttt{(}f \ \overline{e_i}\texttt{)} & \\
 & | & \texttt{(lambda (}\overline{x_i}\texttt{) } e\texttt{)} & \\
 & | & \texttt{(let (}\overline{\texttt{(}x_i \ e_i\texttt{)}}\texttt{) } e\texttt{)} &
\end{array}
\qquad
\begin{array}{rcll}
v & ::= & \texttt{true} & \\
 & | & \texttt{false} & \\
 & | & n & \text{integer} \\
 & | & op & \text{primitive operator} \\
 & | & \mathbf{clo}\,(\overline{x_i}, e, \rho) & \text{closure} \\
 & & & \\
\rho & ::= & \bullet & \\
 & | & (x, v) : \rho &
\end{array}
$$

Figure 9.1: Grammar for TinyScheme.

operational semantics *is not* open to interpretation.

We will re-visit Scheme—first a pure subset, then with support for `set!`—and show how to think carefully and precisely about what language constructs mean using an operational semantics. We will also make a distinction between *expressions* and *values*. In true Scheme, the distinction is fuzzy—many values are valid expressions, including `5` and `(cons 1 2)`—the list containing three items, the symbol `cons`, the number `1`, and the number `2`. However, not all values are valid expressions; for example, the value of a lambda expression is a *compound procedure* or *closure*, which *is not* a valid expression.

The grammar of our pure subset of Scheme, TinyScheme, is given in Figure 9.1. The grammar uses the notation $\overline{e}$ for "one or more e's." For example, when we write $\texttt{(}f \ \overline{e_i}\texttt{)}$, it is equivalent to $\texttt{(}f \ e_1 \ \cdots \ e_n\texttt{)}$. The grammar for expressions should look familiar—it is just a subset of Scheme. We leave the primitive operators, *op*, unspecified. The particular set of primitive operators provided has no effect on the rest of the language, so we leave it as a free parameter of the language.

There are also two new syntactic categories—values, v, and environments, ρ. Values are either constants or a *closure*, $\mathbf{clo}\,(\bar{x}_i, e, \rho)$, which is a pair of code and environment. The code part consist of a list of parameters, $\bar{x}_i$, and an expression, e; the environment is ρ. The Greek letter ρ ("rho") is the standard notation for environments. The grammar for environments in Figure 9.1 tells us that environments can either be empty, $\bullet$ (also standard notation), or a pair of variable and value cons'd to another environment—an environment is just a list of pairs of variables and values.

Our evaluation will take the following form

$$\langle e, \rho \rangle \Downarrow v$$

This can be read "the expression e in the environment ρ evaluates to the value v." We saw this same big-step notation used for the **While** language. We can draw the following analogy between this notation the metacircular evaluator: $\langle e, \rho \rangle \Downarrow v$ is analogous to saying that $\texttt{(mc-eval } e \ \rho\texttt{)}$ returns v. Each operational semantics rule tells us how to evaluate a particular form of TinyScheme expression. The full set of rules is given in Figure 9.2; we will explain them in detail next.

$$\langle \texttt{true}, \rho\rangle \Downarrow \texttt{true} \qquad (\textsc{True})$$

$$\langle \texttt{false}, \rho\rangle \Downarrow \texttt{false} \qquad (\textsc{False})$$

$$\langle n, \rho\rangle \Downarrow n \qquad (\textsc{Integer})$$

$$\langle op, \rho\rangle \Downarrow op \qquad (\textsc{Prim})$$

$$\frac{x \in \rho}{\langle x, \rho\rangle \Downarrow \rho[x]} \qquad (\textsc{Var})$$

$$\frac{\langle e_1, \rho\rangle \Downarrow \texttt{true} \qquad \langle e_2, \rho\rangle \Downarrow v}{\langle \texttt{(if}\ e_1\ \ e_2\ \ e_3\texttt{)}, \rho\rangle \Downarrow v} \qquad (\textsc{If True})$$

$$\frac{\langle e_1, \rho\rangle \Downarrow \texttt{false} \qquad \langle e_3, \rho\rangle \Downarrow v}{\langle \texttt{(if}\ e_1\ \ e_2\ \ e_3\texttt{)}, \rho\rangle \Downarrow v} \qquad (\textsc{If False})$$

$$\langle \texttt{(lambda (}\overline{x_i}\texttt{)}\ \ e\texttt{)}, \rho\rangle \Downarrow \mathbf{clo}\,(\overline{x_i}, e, \rho) \qquad (\textsc{Lambda})$$

$$\frac{\overline{\langle e_i, \rho\rangle \Downarrow v_i} \qquad \langle e, \rho\,[\,\overline{x_i \mapsto v_i}\,]\rangle \Downarrow v}{\left\langle \texttt{(let (}\overline{\texttt{(}x_i\ \ e_i\texttt{)}}\texttt{)}\ \ e\texttt{)}, \rho\right\rangle \Downarrow v} \qquad (\textsc{Let})$$

$$\frac{\langle f, \rho\rangle \Downarrow \mathbf{clo}\,(x_1 \cdots x_n, e, \rho') \qquad \overline{\langle e_i, \rho\rangle \Downarrow v_i} \qquad \langle e, \rho'\,[\,\overline{x_i \mapsto v_i}\,]\rangle \Downarrow v}{\langle \texttt{(}f\ \ e_1 \cdots e_n\texttt{)}, \rho\rangle \Downarrow v} \qquad (\textsc{Apply})$$

$$\frac{\langle f, \rho\rangle \Downarrow op \qquad \overline{\langle e_i, \rho\rangle \Downarrow v_i} \qquad \delta(op, v_1, \ldots, v_n) = v}{\langle \texttt{(}f\ \ e_1 \cdots e_n\texttt{)}, \rho\rangle \Downarrow v} \qquad (\textsc{Apply Prim})$$

Figure 9.2: Big-step rules for evaluating TinyScheme

Evaluating a “Self-Evaluating” Expression

$$\langle \texttt{true}, \rho\rangle \Downarrow \texttt{true} \qquad (\textsc{True})$$

$$\langle \texttt{false}, \rho\rangle \Downarrow \texttt{false} \qquad (\textsc{False})$$

$$\langle n, \rho\rangle \Downarrow n \qquad (\textsc{Integer})$$

$$\langle op, \rho\rangle \Downarrow op \qquad (\textsc{Prim})$$

Integer constants, Boolean constants, and primitives evaluate to themselves in TinyScheme. For constants, this is the behavior we expect based on our experience with Scheme. There is a subtle difference in the way TinyScheme handles primitives—unlike Scheme, which evaluates a primitive operation like + to a “primitive procedure,” TinyScheme evaluates a primitive operation to the primitive itself. That is, Scheme has a special value representation for primitive operations, whereas TinyScheme uses the primitive directly. When we see how primitive operations are applied to arguments, we will see why this representation makes sense.

Evaluating a Variable

In the metacircular evaluator, we evaluate a variable by looking it up in the environment. The rule for TinyScheme does the same

$$\frac{x \in \rho}{\langle x, \rho \rangle \Downarrow \rho[x]} \qquad (\textsc{Var})$$

In order for this rule to "fire," we require that the variable x is in the environment—that's what the notation $x \in \rho$ means. In the metacircular evaluator, if we attempt to look up a variable that isn't in the environment, we get an "Unbound variable" error. The operational semantics doesn't say what to do when a variable is unbound; what happens is up to the implementation. The operational semantics only gives us rules to evaluate expressions that can actually be evaluated!

It would be possible to extend the world of values to encompass errors and then give appropriate rules for handling these errors, but we will not do that for TinyScheme or MiniScheme. If there is not rule to evaluate a given expression, we say evaluation "goes wrong." In this case, the implementation would have to terminate, hopefully with an error that is more informative than "cannot apply any operational semantics rule."

Evaluating an If Expression

$$\frac{\langle e_1, \rho \rangle \Downarrow \texttt{true} \qquad \langle e_2, \rho \rangle \Downarrow v}{\langle \texttt{(if}\ e_1\ e_2\ e_3\texttt{)}, \rho \rangle \Downarrow v} \qquad (\textsc{If True})$$

$$\frac{\langle e_1, \rho \rangle \Downarrow \texttt{false} \qquad \langle e_3, \rho \rangle \Downarrow v}{\langle \texttt{(if}\ e_1\ e_2\ e_3\texttt{)}, \rho \rangle \Downarrow v} \qquad (\textsc{If False})$$

The rules for evaluating if expressions align exactly with what the metacircular evaluator did: evaluate the conditional, and then evaluate either the consequent ("then" branch) or the alternative ("else" branch). There are two rules for evaluating if expressions to cover the separate cases where the conditional evaluates to true or false; these rules are still *deterministic*—we can only ever use at most one of these rules to evaluate an expression.

Evaluating a Lambda Expression

In the environment model of evaluation, when we evaluated a lambda expression, the result was a *compound procedure*, i.e., a pair of code and environment. In the metacircular evaluator, the concrete

$$\langle \texttt{(lambda (}\overline{x_i}\texttt{) } e\texttt{)}, \rho\rangle \Downarrow \mathbf{clo}\,(\overline{x_i}, e, \rho) \tag{Lambda}$$

representation for a compound procedure was a list of four things: the symbol `procedure`, a list of the lambda's parameters, the lambda's body, which is itself a list of expressions, and the current environment at the time the lambda expression was evaluated. For example, the metacircular evaluator would evaluate the lambda expression `(lambda (x) x)` and return `(procedure (x) (x) <env>)`, where `<env>` is the environment passed to the evaluator when it was asked to evaluate the lambda expression.

A closure $\mathbf{clo}\,(\overline{x_i}, e, \rho)$ is just a pair of code and environment! The $\overline{x_i}$ are the parameters of the lambda, the e is the body of the lambda, and ρ is the environment. Using the LAMBDA rule allows us to state the following big-step relation for the lambda expression `(lambda (x) x)`

$$\langle \texttt{(lambda (x) x)}, \rho\rangle \Downarrow \mathbf{clo}\,(x, x, \rho)$$

Evaluating a Let Expression

$$\frac{\overline{\langle e_i, \rho\rangle \Downarrow v_i} \qquad \langle e, \rho\,[\,\overline{x_i \mapsto v_i}\,]\rangle \Downarrow v}{\left\langle \texttt{(let (}\overline{\texttt{(}x_i\ e_i\texttt{)}}\texttt{) } e\texttt{)}, \rho\right\rangle \Downarrow v} \tag{Let}$$

Here are the steps we used to evaluate let expressions in the metacircular evaluator

1. Evaluate the expressions that give the initial values of the variables in the list of bindings.
2. Extend the current environment so that the variables in the let expression's bindings are bound to the values we produced in step 1.
3. Evaluate the body of the let in the new environment.

The first step is is given in the LET rule by the premise $\overline{\langle e_i, \rho\rangle \Downarrow v_i}$. The bar above the premise says that we need to evaluate *every* expression e_i to a value v_i. The notation $\rho\,[\,\overline{x_i \mapsto v_i}\,]$ means an environment that is identical to ρ *except* x_i maps to v_i for all i. This corresponds to step 2. The second premise, $\langle e, \rho\,[\,\overline{x_i \mapsto v_i}\,]\rangle \Downarrow v$, corresponds to step 3—evaluate the body of the let in this new environment. The conclusion states that the entire let expression evaluates to v, the value we got from the second premise.

Unlike an English description of how to evaluate a let expression, the LET rule is unambiguous about the scope of the introduced variables. How do we know what variables are in scope? We look at the environment! The rule is very clear that each variable x_i is only in scope when we evaluate the body of the let; when we evaluate the expressions e_i that give the initial values of the bound variables (in the first premise), we use the environment ρ, which *does not* include any bindings for the variables x_i.

Evaluating an Application

$$\frac{\langle f, \rho \rangle \Downarrow \mathbf{clo}\,(x_1 \cdots x_n, e, \rho') \qquad \overline{\langle e_i, \rho \rangle \Downarrow v_i} \qquad \langle e, \rho' \,[\, \overline{x_i \mapsto v_i} \,] \rangle \Downarrow v}{\langle (f \;\; e_1 \cdots e_n), \rho \rangle \Downarrow v} \quad (\textsc{Apply})$$

$$\frac{\langle f, \rho \rangle \Downarrow op \qquad \overline{\langle e_i, \rho \rangle \Downarrow v_i} \qquad \delta(op, v_1, \ldots, v_n) = v}{\langle (f \;\; e_1 \cdots e_n), \rho \rangle \Downarrow v} \quad (\textsc{Apply Prim})$$

There are two kinds of procedures the metacircular evaluator must be able to apply: primitive procedures, and compound procedures. In TinyScheme's operational semantics, a closure is used to represent a compound procedure, and a primitive operator, *op*, is used to represent a primitive procedure. There are two rules: Apply for applying compound procedures, and Apply Prim for applying primitive procedures. The Apply rule explicitly numbers the arguments and parameters from 1 to n to make it clear that the number of arguments must match the number of parameters.

The metacircular evaluator handles an application by evaluating the operator and operands and then applying the value of the operand to the values of the operators. Both application rules required that the operator be evaluated—this premise has the form $\langle f, \rho \rangle \Downarrow \ldots$ The operands are evaluated via the premise $\overline{\langle e_i, \rho \rangle \Downarrow v_i}$. When applying a closure, the body of the closure is evaluated in a new environment where the parameters—also taken from the closure—are bound to the values of the arguments. When applying a primitive, we appeal to a function δ to compute the result. Using "delta rules" to abstract primitives from the language is a standard technique.

The Apply rule implements *lexical scoping*—a free variable in the body of a lambda is looked up in the environment in which the closure was created. If we instead wanted to implement *dynamic scoping*, we would always look up free variables in the *current* environment. Here is a rule, Apply Dynamic, that implements dynamic scoping

$$\frac{\langle f, \rho \rangle \Downarrow \mathbf{clo}\,(x_1 \cdots x_n, e, \rho') \qquad \overline{\langle e_i, \rho \rangle \Downarrow v_i} \qquad \langle e, \rho \,[\, \overline{x_i \mapsto v_i} \,] \rangle \Downarrow v}{\langle (f \;\; e_1 \cdots e_n), \rho \rangle \Downarrow v} \quad (\textsc{Apply Dynamic})$$

The only difference between this rule and the Apply rule is that we now evaluate the body of the lambda in the current environment, ρ, instead of the environment taken from the closure, ρ'.

We could also decide to evaluate the *arguments* of the application in the environment taken from the closure, as in this useless rule.

$$\frac{\langle f, \rho \rangle \Downarrow \mathbf{clo}\,(x_1 \cdots x_n, e, \rho') \qquad \overline{\langle e_i, \rho' \rangle \Downarrow v_i} \qquad \langle e, \rho' \,[\, \overline{x_i \mapsto v_i} \,] \rangle \Downarrow v}{\langle (f \;\; e_1 \cdots e_n), \rho \rangle \Downarrow v} \quad (\textsc{Useless Apply})$$

Of course if we do this, there is no point in having functions at all—we could have evaluated the "function call" when the function was defined since we never use the current environment!

Finally, let's look at two examples of how to use the rules to evaluate two applications. Here is how we can use the rules to evaluate `((lambda (x) x) 1)`

$$\frac{\langle \texttt{(lambda (x) x)}, \rho\rangle \Downarrow \textbf{clo}\,(x, x, \rho) \qquad \langle \texttt{1}, \rho\rangle \Downarrow 1 \qquad \langle \texttt{x}, \rho\,[x \mapsto 1]\rangle \Downarrow 1}{\langle \texttt{((lambda (x) x) 1)}, \rho\rangle \Downarrow 1}\ (\textsc{Apply})$$

Here is how we can use the rules to evaluate `(+ 1 2)`

$$\frac{\langle \texttt{+}, \rho\rangle \Downarrow + \qquad \langle \texttt{1}, \rho\rangle \Downarrow 1 \qquad \langle \texttt{2}, \rho\rangle \Downarrow 1 \qquad \delta(+, 1, 2) = 3}{\langle \texttt{(+ 1 2)}, \rho\rangle \Downarrow 3}\ (\textsc{Apply Prim})$$

9.1 Adding Support for State

TinyScheme is pure—it has no side effects! How can we extend this language to handle state? Recall that in the substitution model of evaluation, variables stood for *values*—just like the environment ρ in our operational semantics for TinyScheme. We used a new model of evaluation to handle Scheme with state—the environment model of evaluation. In this model of evaluation, variables no longer stood for values, but for *locations*. Looking up the value of a variable required reading from this location, and modifying the variable required writing to this location.

We will make a similar change to our operational semantics to add support for state. Instead of mapping variables to values, our environment ρ will map variables to locations. How do we keep track of locations? We will use a *store*, which maps locations to values. You can think of the locations as "pointers" and store as "memory." In this analogy, the environment maps variables to pointers; to find the value of a variable, we look up the pointer and dereference it (by accessing the store); to change the value of a variable, we look up the pointer and write a new value to it (again, by accessing the store).

To handle a store, our evaluation relation must change. The new relation has this form

$$\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$$

To evaluate an expression, we need an environment, ρ, to look up variables' locations, and a store, σ, to find the value at a location or overwrite that value. The result of evaluation is a value *and a new store*. The new store contains any changes made during the evaluation of e. Why can't we just change the original store "in-place?" Because we're using the pure language of math to describe our operational semantics, the only option we have to model state is to take the current state as "input" and provide the altered state as "output." This is exactly what we did to express the operational semantics of statements in the **While** language.

Creating a binding for a variable x to a value v is now more complicated because we need to find a *location* so we can store the value of the variable. We will use "script ell," ℓ, to refer to locations in our semantics. Instead of modeling a memory allocator in our rules, we will just require a "fresh" ℓ—a location that is not being used. We leave the details of how to implement allocation to the implementation.

Before we had to support state, we created a new binding for a variable by writing $\rho' = \rho\,[x \mapsto v]$. Now, we need to perform multiple steps

1. Find a fresh location ℓ.
2. Create a new environment that maps the variable to this fresh location, $\rho' = \rho\,[x \mapsto \ell]$.
3. Update the sore so that this fresh location is mapped to the value of the variable, $\sigma' = \sigma\,[\ell \mapsto v]$.

Evaluating a Variable

$$\frac{x \in \rho \qquad \ell = \rho[x] \qquad \ell \in \sigma}{\langle x, \rho, \sigma\rangle \Downarrow \langle \sigma[\ell], \sigma\rangle} \tag{\textsc{Var}}$$

When evaluating a variable, we now must first look up the variable's location in the environment, ρ, and then we can find the value associated with that location in the store, σ. Note that this rule only applies when the variable exists in the environment and its associated location exists in the store. Although we have not yet seen the rest of the rules, when we add a variable to the environment, we also always add its associated location to the store. This means the implementation should not have to check that the location actually exists when we look up the variables value. Nonetheless, we put the check in the rule to make it clear that the location must exist in the store for us to be able to dereference it.

Evaluating `set!`

$$\frac{\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle \qquad x \in \rho \qquad \ell = \rho[x] \qquad \ell \in \sigma'}{\langle \texttt{(set!}\ \ x\ \ e\texttt{)}, \rho, \sigma\rangle \Downarrow \langle \texttt{true}, \sigma'\,[\ell \mapsto v]\rangle} \tag{\textsc{Set}}$$

Evaluating `set!` requires us to find the location of the variable whose value we are modifying just as the VAR rule did. Note that evaluating the expression that gives the new value of the variable x could change the environment, so we must update the value in this (potentially) modified environment. In the metacircular evaluator, the value of a `set!` expression was the symbol `ok`; in MiniScheme, the value of a `set!` expression is `true`.

Evaluating a Let Expression

$$\frac{\overline{\langle e_i, \rho, \sigma_{i-1}\rangle \Downarrow \langle v_i, \sigma_i\rangle} \qquad \left\langle body, \rho\left[\,\overline{x_i \mapsto \ell_i}\,\right], \sigma_n\left[\,\overline{\ell_i \mapsto v_i}\,\right]\right\rangle \Downarrow_B \langle v, \sigma'\rangle \qquad \overline{\ell_i}\quad \text{fresh}}{\left\langle \texttt{(let (}\overline{\texttt{(}x_i\ \ e_i\texttt{)}}\texttt{)}\ \ body\texttt{)}, \rho, \sigma_0\right\rangle \Downarrow \langle v, \sigma'\rangle} \tag{\textsc{Let}}$$

When evaluating a let expression, we now must account for the possibility that when we evaluate the expressions that give the values of the bound variables, the store could be modified.

We still evaluate these expressions using the current environment, but now we have to "thread" the state through the evaluation of each expression. The conclusion of the rule states that the initial state is σ_0. The first premise, $\langle e_i, \rho, \sigma_{i-1}\rangle \Downarrow \langle v_i, \sigma_i\rangle$, says that to evaluate e_i, we use the store σ_{i-1} and return a new store σ_i. For example, to evaluate e_1, we use σ_0 and return σ_1. Therefore σ_n is the store we have in hand after evaluating all the expressions that give the initial values of the variables. We use this store, σ_n, to evaluate the body of the let.

Evaluating Other Expressions

The full set of evaluation rules for expressions is given in Figure 9.3. The other forms of expressions are evaluated as they were in TinyScheme, except now we must thread the store through each evaluation step.

New MiniScheme Language Constructs

As well as `set!`, MiniScheme supports several constructs that are familiar from Scheme but that were not present in TinyScheme: `define`, `begin`, and let and lambda bodies that are *sequences* of definitions and expressions rather than a single expression. The complete grammar for MiniScheme is given in Figure 9.5; here we give the changes relative to TinyScheme

A *body* is a sequence of definitions or expressions followed by a final expression. The intuition is that the value of the final expression will be the value of the *body*. Why do we need a list of definitions *and* expressions? Because we may want to call some functions with side effects or use `set!` before we evaluate the final expression. In MiniScheme, a *body* is like a sequence of statements in C, except that we can also use local definitions along the way.

There is an issue we need to handle when evaluating definitions. Consider this code

```
(begin (define x 1)
       (define y 2)
       (+ x y))
```

It is clear that this `begin` expression should evaluate to `3`. That is because the variables `x` and `y` scope over *all following definitions*. To reason about what a single `define` does, we need to know about *all following definitions*—we can't reason locally about a single `define`.

To contrast, consider this code

```
(let ((x 1)
      (y 2))
  (+ x y))
```

Here, the variables `x` and `y` scope only over the body of the `let`—we can reason about the let locally since the variables it binds go out of scope once the body of the let has been evaluated. The evaluation rule for a let is *compositional* because we only need to reason locally to determine the

$$\langle \mathtt{true}, \rho, \sigma\rangle \Downarrow \langle \mathtt{true}, \sigma\rangle \qquad (\textsc{True})$$

$$\langle \mathtt{false}, \rho, \sigma\rangle \Downarrow \langle \mathtt{false}, \sigma\rangle \qquad (\textsc{False})$$

$$\langle n, \rho, \sigma\rangle \Downarrow \langle n, \sigma\rangle \qquad (\textsc{Integer})$$

$$\langle op, \rho, \sigma\rangle \Downarrow \langle op, \sigma\rangle \qquad (\textsc{Prim})$$

$$\frac{x \in \rho \qquad \ell = \rho[x] \qquad \ell \in \sigma}{\langle x, \rho, \sigma\rangle \Downarrow \langle \sigma[\ell], \sigma\rangle} \qquad (\textsc{Var})$$

$$\frac{\langle e_1, \rho, \sigma\rangle \Downarrow \langle \mathtt{true}, \sigma'\rangle \qquad \langle e_2, \rho, \sigma'\rangle \Downarrow \langle v, \sigma''\rangle}{\langle (\mathtt{if}\ \ e_1\ \ e_2\ \ e_3), \rho\rangle \Downarrow \langle v, \sigma''\rangle} \qquad (\textsc{If True})$$

$$\frac{\langle e_1, \rho, \sigma\rangle \Downarrow \langle \mathtt{false}, \sigma'\rangle \qquad \langle e_3, \rho, \sigma'\rangle \Downarrow \langle v, \sigma''\rangle}{\langle (\mathtt{if}\ \ e_1\ \ e_2\ \ e_3), \rho\rangle \Downarrow \langle v, \sigma''\rangle} \qquad (\textsc{If False})$$

$$\langle (\mathtt{lambda}\ \ (\overline{x_i})\ \ body), \rho, \sigma\rangle \Downarrow \langle \mathbf{clo}\left(\overline{x_i}, body, \rho\right), \sigma\rangle \qquad (\textsc{Lambda})$$

$$\frac{\overline{\langle e_i, \rho, \sigma_{i-1}\rangle \Downarrow \langle v_i, \sigma_i\rangle} \qquad \left\langle body, \rho\left[\overline{x_i \mapsto \ell_i}\right], \sigma_n\left[\overline{\ell_i \mapsto v_i}\right]\right\rangle \Downarrow_B \langle v, \sigma'\rangle \qquad \overline{\ell_i} \quad \text{fresh}}{\left\langle (\mathtt{let}\ \ (\overline{(x_i\ \ e_i)})\ \ body), \rho, \sigma_0\right\rangle \Downarrow \langle v, \sigma'\rangle} \qquad (\textsc{Let})$$

$$\frac{\begin{array}{c}\langle f, \rho, \sigma\rangle \Downarrow \langle \mathbf{clo}\left(\overline{x_i}, body, \rho'\right), \sigma_0\rangle \qquad \overline{\langle e_i, \rho, \sigma_{i-1}\rangle \Downarrow \langle v_i, \sigma_i\rangle} \qquad \overline{\ell_i} \quad \text{fresh} \\ \left\langle body, \rho\left[\overline{x_i \mapsto \ell_i}\right], \sigma_n\left[\overline{\ell_i \mapsto v_i}\right]\right\rangle \Downarrow_B \langle v, \sigma'\rangle\end{array}}{\langle (f\ \ e_1 \cdots e_n), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \qquad (\textsc{Apply})$$

$$\frac{\langle f, \rho, \sigma\rangle \Downarrow \langle op, \sigma_0\rangle \qquad \overline{\langle e_i, \rho, \sigma_{i-1}\rangle \Downarrow \langle v_i, \sigma_i\rangle} \qquad \delta(op, v_1, \ldots, v_n) = v}{\langle (f\ \ e_1 \cdots e_n), \rho, \sigma\rangle \Downarrow \langle v, \sigma_n\rangle} \qquad (\textsc{Apply Prim})$$

$$\frac{\langle body, \rho, \sigma\rangle \Downarrow_B \langle v, \sigma'\rangle}{\langle (\mathtt{begin}\ \ body), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \qquad (\textsc{Begin})$$

$$\frac{\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle \qquad x \in \rho \qquad \ell = \rho[x] \qquad \ell \in \sigma'}{\langle (\mathtt{set!}\ \ x\ \ e), \rho, \sigma\rangle \Downarrow \langle \mathtt{true}, \sigma'\left[\ell \mapsto v\right]\rangle} \qquad (\textsc{Set})$$

$$\frac{\langle e_i, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle (\mathtt{amb}\ \ \overline{e_i}), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \qquad (\textsc{Amb})$$

Figure 9.3: Big-step rules for evaluating MiniScheme expressions.

$$\begin{array}{lcl}
decl & ::= & \texttt{(define } x\ body\texttt{)} \\
 & | & \texttt{(define (}f\ \overline{x_i}\texttt{)}\ body\texttt{)} \\
 & | & e \\
 & & \\
body & ::= & \overline{decl}\ e \\
 & & \\
e & ::= & \ldots \\
 & | & \texttt{(lambda (}\overline{x_i}\texttt{)}\ body\texttt{)} \\
 & | & \texttt{(let (}\overline{\texttt{(}x_i\ e_i\texttt{)}}\texttt{)}\ body\texttt{)} \\
 & | & \texttt{(begin}\ body\texttt{)} \\
 & | & \texttt{(set!}\ x\ e\texttt{)}
\end{array}$$

Figure 9.4: Grammar changes from TinyScheme to MiniScheme.

effect of the let. In contrast, the rule for evaluating `define` must be *non-compositional* because we cannot reason about it locally—we have to know the context in which the `define` appears. Therefore, instead of giving a rule to evaluate a single declaration, we will give a rule to evaluate a *list* of declarations.

We will use two new evaluation relations to evaluate bodies and lists of declarations

$$\left\langle \overline{decl}, \rho, \sigma \right\rangle \Downarrow_D \langle \rho', \sigma' \rangle$$
$$\langle body, \rho, \sigma \rangle \Downarrow_B \langle v, \sigma' \rangle$$

Unlike the evaluation relations for expressions or bodies, the evaluation relation for declarations takes an environment as input *and* produces and environment as output. This is how we ensure that definitions scope over all subsequent declarations.

The rule for evaluating bodies is simple: we evaluate the list of declarations in the body and then use the resulting environment and store to evaluate the final expression in the body

$$\frac{\left\langle \overline{decl}, \rho, \sigma \right\rangle \Downarrow_D \langle \rho', \sigma' \rangle \qquad \langle e, \rho', \sigma' \rangle \Downarrow \langle v, \sigma'' \rangle}{\left\langle \overline{decl}\ e, \rho, \sigma \right\rangle \Downarrow_B \langle v, \sigma'' \rangle} \quad \text{(Body)}$$

Evaluating a list of declarations requires four rules; one for the base case (empty list), and one each for the three types of declarations: “regular” definitions, function definitions, and expressions. The rule for evaluating an empty list of declarations is trivial: we pass the environment and state through unchanged

$$\langle \bullet, \rho, \sigma \rangle \Downarrow_D \langle \rho, \sigma \rangle \quad \text{(Empty)}$$

$$
\begin{array}{lcl}
\mathit{decl} & ::= & \texttt{(define } x\ \mathit{body}\texttt{)} \\
 & | & \texttt{(define (}f\ \overline{x_i}\texttt{)}\ \mathit{body}\texttt{)} \\
 & | & e \\
 & & \\
\mathit{body} & ::= & \overline{\mathit{decl}}\ e \\
 & & \\
e & ::= & \texttt{true} \\
 & | & \texttt{false} \\
 & | & n \\
 & | & \mathit{op} \\
 & | & x \\
 & | & \texttt{(if } e\ e\ e\texttt{)} \\
 & | & \texttt{(}f\ \overline{e_i}\texttt{)} \\
 & | & \texttt{(lambda (}\overline{x_i}\texttt{)}\ \mathit{body}\texttt{)} \\
 & | & \texttt{(let (}\overline{\texttt{(}x_i\ e_i\texttt{)}}\texttt{)}\ \mathit{body}\texttt{)} \\
 & | & \texttt{(begin } \mathit{body}\texttt{)} \\
 & | & \texttt{(set! } x\ e\texttt{)} \\
 & | & \texttt{(amb } \overline{e_i}\texttt{)}
\end{array}
\qquad
\begin{array}{lcl}
v & ::= & \texttt{true} \\
 & | & \texttt{false} \\
 & | & n \\
 & | & \mathit{op} \\
 & | & \mathbf{clo}\,(\overline{x_i}, \mathit{body}, \rho) \\
 & & \\
\rho & ::= & \bullet \\
 & | & (x, \ell) : \rho \\
 & & \\
\sigma & ::= & \bullet \\
 & | & (\ell, v) : \sigma
\end{array}
$$

Figure 9.5: Grammar for MiniScheme.

To evaluate a definition, we first evaluate the body of the definition. Then we must allocate a fresh location, ℓ, and create a new environment where the variable x maps to this location. Finally, we evaluate the following declarations in this new environment and a store updated so that ℓ maps to the value of the body of the definition.

$$
\frac{\langle \mathit{body}, \rho, \sigma\rangle \Downarrow_B \langle v, \sigma'\rangle \qquad \ell \text{ fresh} \qquad \rho' = \rho\,[x \mapsto \ell] \qquad \left\langle \overline{\mathit{decl}}, \rho', \sigma'\,[\ell \mapsto v]\right\rangle \Downarrow_D \langle \rho'', \sigma''\rangle}{\left\langle \texttt{(define } x\ \mathit{body}\texttt{)}\ \overline{\mathit{decl}}, \rho, \sigma\right\rangle \Downarrow_D \langle \rho'', \sigma''\rangle}\ (\textsc{Define})
$$

Why do we need two forms of `define`? Consider the following definition of the recursive factorial function

```
(define fact
  (lambda (n) (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
```

Is `fact` in scope in the body of the lambda? Looking at the rules, we can see that the answer is no—the environment used to evaluate the body of the definition is the same environment we were asked to use to evaluate the entire definition itself. It's possible that *some* variable fact is in scope over the body of the definition, but if so, it would be a *prior* definition. For example, in the following code, the variable fact in the body of the lambda would refer to the *first* definition of fact, whose value is 1, not the second definition.

```
(define fact 1)
(define fact
  (lambda (n) (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
```

To solve this problem, we need to make sure that when we create the closure for the lambda expression, the environment in the closure has a reference to the function we are defining. Concretely, when we evaluate the lambda expression that is the factorial function, we must create a closure whose environment ρ somehow points back to the closure itself. How can we do this? Remember that environment no longer map variables to values, but to *locations*. This gives us a way to create a "self-referential" closure

1. Create a fresh location, ℓ, that will point to the value of the closure.
2. Create a new environment, ρ', where fact maps to ℓ. Notice that we have not yet mapped this location to anything in the store; it is a "dangling reference."
3. Evaluate the lambda expression *using the new environment* to produce the closure.
4. Evaluate the rest of the declarations in a new store where ℓ maps to the closure we just created.

This is exactly what the Define Fun rule does

$$\frac{\langle \texttt{(lambda (}\overline{x_i}\texttt{)}\ body\texttt{)}, \rho', \sigma\rangle \Downarrow \langle v, \sigma'\rangle \qquad \ell \text{ fresh} \qquad \rho' = \rho\,[f \mapsto \ell] \\ \left\langle \overline{decl}, \rho', \sigma'\,[\ell \mapsto v]\right\rangle \Downarrow_D \langle \rho'', \sigma''\rangle}{\left\langle \texttt{(define (}f\ \overline{x_i}\texttt{)}\ body\texttt{)}\ \overline{decl}, \rho, \sigma\right\rangle \Downarrow_D \langle \rho'', \sigma''\rangle}\ (\textsc{Define Fun})$$

This rule appears to be identical to the standard lambda-desugaring for the function variant of define. The main difference is that we use ρ', which contains a mapping from f to our "dangling pointer" ℓ, to evaluate the lambda. Although we have named the result of evaluating the lambda expression v, we know that this value must take the form of a closure. Because we evaluated the lambda expression with ρ', this closure will also contain ρ'.

Why is it "safe" to have a "dangling pointer" like this? When we evaluate a lambda expression, *we don't actually evaluate the body of the lambda.* The body of the lambda is only evaluated if the function is called. That cannot happen until we evaluate some of the following declarations, and

$$\langle \bullet, \rho, \sigma\rangle \Downarrow_D \langle \rho, \sigma\rangle \qquad \text{(Empty)}$$

$$\frac{\langle \mathit{body}, \rho, \sigma\rangle \Downarrow_B \langle v, \sigma'\rangle \qquad \ell \text{ fresh} \qquad \rho' = \rho[x \mapsto \ell] \qquad \left\langle \overline{\mathit{decl}}, \rho', \sigma'[\ell \mapsto v]\right\rangle \Downarrow_D \langle \rho'', \sigma''\rangle}{\left\langle \texttt{(define } x\ \mathit{body}\texttt{)}\ \overline{\mathit{decl}}, \rho, \sigma\right\rangle \Downarrow_D \langle \rho'', \sigma''\rangle} \qquad \text{(Define)}$$

$$\frac{\langle \texttt{(lambda (}\overline{x_i}\texttt{)}\ \mathit{body}\texttt{)}, \rho', \sigma\rangle \Downarrow \langle v, \sigma'\rangle \qquad \ell \text{ fresh} \qquad \rho' = \rho[f \mapsto \ell] \qquad \left\langle \overline{\mathit{decl}}, \rho', \sigma'[\ell \mapsto v]\right\rangle \Downarrow_D \langle \rho'', \sigma''\rangle}{\left\langle \texttt{(define (}f\ \overline{x_i}\texttt{)}\ \mathit{body}\texttt{)}\ \overline{\mathit{decl}}, \rho, \sigma\right\rangle \Downarrow_D \langle \rho'', \sigma''\rangle} \qquad \text{(Define Fun)}$$

$$\frac{\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\left\langle e\ \overline{\mathit{decl}}, \rho, \sigma\right\rangle \Downarrow_D \langle \rho, \sigma'\rangle} \qquad \text{(Exp)}$$

$$\frac{\left\langle \overline{\mathit{decl}}, \rho, \sigma\right\rangle \Downarrow_D \langle \rho', \sigma'\rangle \qquad \langle e, \rho', \sigma'\rangle \Downarrow \langle v, \sigma''\rangle}{\left\langle \overline{\mathit{decl}}\ e, \rho, \sigma\right\rangle \Downarrow_B \langle v, \sigma''\rangle} \qquad \text{(Body)}$$

Figure 9.6: Big-step rules for evaluating MiniScheme declarations and bodies.

by that time, we have updated the store so that ℓ points to the function's closure—it will no longer be dangling.

The final rule for evaluating expressions that occur in a list of declarations is straightforward: evaluate the expression and pass through the environment unchanged.

$$\frac{\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\left\langle e\ \overline{\mathit{decl}}, \rho, \sigma\right\rangle \Downarrow_D \langle \rho, \sigma'\rangle} \qquad \text{(Exp)}$$

The rules for evaluating declarations and bodies are collected in Figure 9.6.

9.2 Implementing MiniScheme

The mapping between concepts from the operational semantics and the Haskell implementation is given in Table 9.1. All syntactic categories (expressions, declarations, bodies) as well as environments and stores are represented using Haskell data types. The three evaluation relations are represented as Haskell functions.

The declarations for the data types representing locations, environments, and stores is as follows

Semantics	**Haskell**
e	`Exp`
v	`Val`
$body$	`Body`
$decl$	`Decl`
$\overline{decl}$	`[Decl]`
ℓ	`Loc`
ρ	`Env`
σ	`State`
$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$	`eval`
$\left\langle \overline{decl}, \rho, \sigma \right\rangle \Downarrow_D \langle \rho', \sigma' \rangle$	`evalDecls`
$\langle body, \rho, \sigma \rangle \Downarrow_B \langle v, \sigma' \rangle$	`evalBody`

Table 9.1: Mapping between MiniScheme operational semantics and the Haskell implementation.

```haskell
type Loc = Int

type Env = Map Var Loc

data State = State { nextLoc :: !Loc
                   , heap    :: Map Loc Val
                   }
```

A location is just an integer. An environment is a map from variables to locations—a `Map` is the standard Haskell "dictionary" data type (see LYAH, Chapter 7). The store, represented by the `State` data type, contains both a map from locations to values, as we would expect, as well as a location, `nextLoc`. To allocate a new location, we will just increment this counter—we never worry about garbage collection! The "bang" on the `nextLoc` field's type is a strictness annotation (if you are overly curious, see *Real World Haskell*, Chapter 25).

In the operational semantics, we had to manually pass around the environment and store (ρ and σ, respectively). Fortunately, we don't have to do that in the Haskell implementation because we can leverage monads to do the "threading" for us. In fact, monads were originally used to describe the semantics of programming languages. Our evaluation relation for expressions takes an environment as input, which matches the `MonadReader` pattern. It also takes a store as input *and* produces an updated store as output, which matches the `MonadState` pattern. We therefore define a new type class to capture the constraints we want our evaluation monad to have

```
class (MonadPlus m,
       MonadReader Env m,
       MonadState State m,
       MonadIO m) => MonadEval m where
```

The `MonadIO` constraint will allow us to perform `IO` actions during evaluation.[1] The `MonadPlus` constraint abstracts over monads that allow us to express choice and failure (see LYAH, Chapter 13)—this will allow us to implement `amb`.

[1]Unfortunately we have to use the `liftIO` function to "lift" `IO` actions into our monad (see *Real World Haskell*, Chapter 15).