A Term Rewriting Model for Scheme
I. The Functional Kernel

1 Rewrite Rules

1.1 Primitive Application Rules

A useful way to think about computation in Scheme is as *algebraic simplification* aimed at computing the *value* of an expression.

Let's start by considering a simple arithmetic combination of numbers in Scheme notation:

```
(+ 0 1 (- 2 3) (* 4 5 (* -6 -7))).
```

We take for granted an infinite set of *primitive application* rules for rewriting such expressions, for example,

$$\begin{aligned}\texttt{(- 2 3)} &\longrightarrow \texttt{-1}\\ \texttt{(* 4 5 42)} &\longrightarrow \texttt{840}\\ &\vdots\end{aligned}$$

In general, a primitive application rule is of the form

$$(\langle\text{primitive procedure id}\rangle\ \langle\text{primitive value}_1\rangle \ldots) \longrightarrow \langle\text{primitive value}\rangle,$$

where ⟨primitive value⟩'s are atomic expressions like numbers or truth values, and primitive procedure identifiers are names like `+`, `<`, and `atan`, etc., which in Scheme are bound to "built-in" procedures applicable to primitive values.

By repeatedly replacing left hand sides of rules with their right hand sides, we can eventually rewrite the expression above to its numerical value. As an aid in reading the calculation, we underline the successive subexpressions which are to be rewritten:

$$\begin{aligned}\texttt{(+ 0 1 (- 2 3) (* 4 5 }\underline{\texttt{(* -6 -7)}}\texttt{))} &\longrightarrow \texttt{(+ 0 1 }\underline{\texttt{(- 2 3)}}\texttt{ (* 4 5 42))}\\ &\longrightarrow \texttt{(+ 0 1 -1 }\underline{\texttt{(* 4 5 42)}}\texttt{)}\\ &\longrightarrow \texttt{(+ 0 1 -1 840)}\\ &\longrightarrow \texttt{840}\end{aligned}$$

Another familiar set of primitive application rules result in the *Booleans* `#t`, `#f`:

$$\begin{aligned}\texttt{(< 6 7)} &\longrightarrow \texttt{\#t}\\ \texttt{(zero? -1.414)} &\longrightarrow \texttt{\#f}\\ &\vdots\end{aligned}$$

For example, using these rules and the numerical ones above allows us to handle:

$$\begin{aligned}
\texttt{(< (+ 0 1 2) }\underline{\texttt{(/ 3 1)}}\texttt{)} &\longrightarrow \texttt{(< }\underline{\texttt{(+ 0 1 2)}}\texttt{ 3)} \\
&\longrightarrow \texttt{(< 3 3)} \\
&\longrightarrow \texttt{\#f}
\end{aligned}$$

Note that there is no specified order or location for applying the rules. We might have chosen to rewrite the example above using the rules in another order:

$$\begin{aligned}
\texttt{(< }\underline{\texttt{(+ 0 1 2)}}\texttt{ (/ 3 1))} &\longrightarrow \texttt{(< 3 }\underline{\texttt{(/ 3 1)}}\texttt{)} \\
&\longrightarrow \texttt{(< 3 3)} \\
&\longrightarrow \texttt{\#f}
\end{aligned}$$

An expression to which no rules can be applied is called *terminated*. Primitive rewriting has two familiar, important properties:

- **Termination**. No matter in what order the rewriting rules are applied, a terminated expression[1] will eventually be reached.
- **Uniqueness**. No matter in what order the rewriting rules are applied, the terminated expression arrived at is the same!

The Termination property is obvious because primitive applications always shorten an expression. We'll say more about Uniqueness below.

1.2 Immediate Values

In addition to the primitive values, procedure identifiers like `+` or `<` are themselves a kind of value[2]. Scheme also provides a means to *construct* procedures values, namely, with *lambda expressions*. The

- primitive values,
- procedure identifiers, and
- lambda expressions,

are called *immediate values.*

[1] We usually expect the terminated expression to be a primitive value, but some "ill-typed" expressions like `(+ #t 3)` are terminated but do not have values. (What would Scheme do when such an expression is evaluated?)

[2] We assume here that procedure identifiers *won't* be redefined! Note that according to the official definition of Revised^4 Scheme, these identifiers *are* treated as variables and it is possible to redefine them. As a matter of fact, in 6.001 Scheme, the arithmetic operators do get redefined slightly from general MIT Scheme—so that novice users don't unexpectedly generate the huge quotients possible from Scheme's infinite-precision rational number operations.

1.3 Conditional Rules

Other familiar algebraic rules like

$$\begin{aligned}(1 \times A) &= A\\(A + 0) &= A\\(A \times 0) &= 0\\&\vdots\end{aligned}$$

allow simplifications of arbitrary subexpressions, A, which may contain variables, as opposed to just combinations of constants as above.

Scheme's rules for `if` expressions, also called *conditionals*, resemble this sort of algebraic rule. There are only two rewrite rules for an `if` expression:

`(if #f` $\langle\text{expression}_1\rangle$ $\langle\text{expression}_2\rangle$`)` $\longrightarrow$ $\langle\text{expression}_2\rangle$,
`(if` V $\langle\text{expression}_1\rangle$ $\langle\text{expression}_2\rangle$`)` $\longrightarrow$ $\langle\text{expression}_1\rangle$,
when V is an $\langle$immediate value$\rangle$ not equal to `#f`.

Simple as they are, these two rules differ significantly from the primitive application rules because $\langle\text{expression}_1\rangle$ and $\langle\text{expression}_2\rangle$ may be *any* Scheme expressions, not just immediate values as with primitive applications.

Now we can do rewriting such as

```
   (if (lambda (a b) c) (if (< (+ 0 1 2) (/ 3 1)) atan (lambda (x) x)) +)
⟶ (if (< (+ 0 1 2) (/ 3 1)) atan (lambda (x) x))
     ⋮
⟶ (if #f atan (lambda (x) x))
⟶ (lambda (x) x)
```

As long as we restrict to a *functional* subset of Scheme[3], i.e., Scheme without side-effects, the termination and uniqueness properties still hold for rewritings using both primitive applications and conditional rules. In particular, although rewriting the second or third expressions (called the *consequent* and the *alternative* of the conditional) before simplifying the first expression (called the *test*) may waste some effort, or may even cause an error, it will never result in a different value than the obvious strategy of first rewriting the test into an immediate value.

Problem 1. The algebraic rules above do *not* accurately reflect Scheme behavior.

(a) Describe a definition of `x` such that evaluation of `(* 1 x)` does not return the value of `x`.

(b) Describe a simple Scheme expression, A, such that evaluation of `(*` A `0)` neither returns `0` nor interrupts with an error.

[3]Appendix A contains a grammar for the functional part of Scheme we consider.

2 Substitution Models

Another familiar part of algebra is the use of names for expressions, for example, the names "p," "p_1," and "p_2" in:

> Let $p(x) = x^2 - 1$. The polynomial p factors into $p_1 \cdot p_2$, where $p_1(x) = x + 1$ and $p_2(x) = x - 1$.

Variable instantiation is a basic naming rule of algebra: substitute an expression for an occurrence of its name. For example, using instantiation and primitive applications in Scheme, we can carry out the following calculation:

```
(define y 7) (define x <) (x (+ 1 2) y)
    ⟶ (define y 7) (define x <) (x (+ 1 2) 7)
    ⟶ (define y 7) (define x <) (x 3 7)
    ⟶ (define y 7) (define x <) (< 3 7)
    ⟶ (define y 7) (define x 3) #t
```

A system of rules which explain Scheme computation in terms of symbolic manipulation of Scheme expressions and definitions is called a *Substitution Model* for Scheme.

For ordinary algebra, naming rules are not much emphasized. In contrast, for nearly all programming languages, rules for handling names and their definitions play a key role—arguably a more important one than the ordinary algebraic simplification rules described in Section 1.

2.1 Instantiation

The Scheme version of the *variable instantiation rule* is:

$$\langle\text{variable}\rangle \longrightarrow \langle\text{immediate value}\rangle$$

> when ⟨variable⟩ is bound to ⟨immediate value⟩ and no variable binding is changed by the rewriting.

There are three aspects of this rule which need to be explained.

First, we need to specify which, if any, definitions *bind* which occurrences of variables, in order to know which immediate value to substitute where—after all, there may be different definitions of the same variable in different parts of an expression. Scheme's binding rule is quite simple: starting at the occurrence of the variable to be instantiated, search back through the enclosing subexpressions until arriving at a definition of that variable.[4]

[4] A more precise specification of Scheme's binding (also called *scoping*) rules, is given in Appendix B.

Second, we note that variables are only instantiated with *immediate values*, not arbitrary expressions, because in our kernel Scheme, definitions may only bind variables to immediate values. For example, a definition such as

```
(define y (+ 1 2)) (* y y)
```

is not part of the functional kernel.[5]

Third is the condition that variable binding not be changed by instantiation. For example, in Scheme we could define `add2` and `add3` procedures as follows:

```
(define n 2)
(define add2 (lambda (m) (+ m n)))
(define add3 (lambda (n) (+ 1 (add2 n))))
```

so that evaluating `(add3 5)` would return `8`.

But suppose we **naively** instantiate the underlined occurrence of `add2`, obtaining:

```
(define n 2)
(define add2 (lambda (m) (+ m n)))
(define add3 (lambda (n) (+ 1 ((lambda (m) (+ m n)) n))))
```

Now, evaluating `(add3 5)` would return `11`!

The problem is that the occurrence of `n` is bound to `2` in the definition of `add2`, but in the instantiated copy of `add2`'s value, the copied occurrence of `n` has become bound by the `lambda (n)` in the definition of `add3`. This change in `n`'s binding is what has to be prevented in order to reflect Scheme's static scoping discipline.

Instantiation at arbitrary positions will be possible after we describe some additional rules in Section 2.2.2 and Appendix B for changing defined names into "fresh" names. However, there is no need for renaming, and no problem about changed bindings, as long as we obey

> **The Lambda Body Rewriting Restriction**: *Don't rewrite any expression within the body of a lambda expression.*

This works because in Scheme grammar, the only place definitions can occur is at "top level" or at the beginning of lambda bodies. So there is no possibility of bindings being changed by instantiating occurrences of variables outside lambda bodies. Real Scheme evaluates in an order obeying the lambda body restriction.

Problem 2. Describe a simple Scheme expression, M, such that evaluation of M returns 0 but evaluation of N returns 1, where N is the expression which results from a naive instantiation (without renaming) inside some lambda body in M.

[5] Such non-immediate definitions will be handled later when we extend our model to include assignment statements, i.e., Scheme's special form `set!`.

2.2 Lambda Applications

The Substitution Model sketched in 6.001 handled procedure applications with a *substitution rule* rule of the form:

> Rewrite "`((lambda (`x`)` B`)` V`)`" as "B with V substituted for x,"
> where V is an immediate value.

In these notes we describe a slightly different rule for procedure application. Using this new rule dramatically shortens the size of rewritten expressions in many examples. It also will allow us to model Scheme's computational behavior more accurately.[6]

We'll simply call the new rule the *lambda application rule*. A simplified example of this rule is:

> Rewrite "`((lambda (`x`)` B`)` V`)`" as "B,"
> while adding the definition "`(define` x V`)`."

Of course to use this rule, we'll need to specify where and how to "add" definitions.

It's helpful to realize that the new lambda application rule could be used to simulate the old 6.001 rule: starting with `((lambda (`x`)` B`)` V`)`, use the lambda application rule. This gives B along with the definition `(define` x V`)`. Then, by successively instantiating all the x's in B by copies of V, we arrive at the same result as the substitution rule. Well almost—we still have the definition of x hanging around. But since we instantiated all the x's, this definition no longer refers to anything and is not needed. So we will apply an additional "garbage collection" rule which allows us to remove unreferenced definitions.

2.2.1 Bodies

Note that we might have tried expressing the rule above as

$$\begin{aligned}&\texttt{((lambda (}x\texttt{)}\ B\texttt{)}\ V\texttt{)}\\ &\qquad\longrightarrow \texttt{(define}\ x\ V\texttt{)}\ B.\end{aligned}$$

This version introduces a slight grammatical complication: the expression on the left hand side of the rule rewrites into a sequence of two things—a definition followed by the body, B, of the lambda expression. If there were several formal parameters instead of just x, there would be several definitions preceding B. What's more, B itself might be a sequence of definitions followed by an expression.

[6] The Substitution Model in 6.001 is sketchily described (on purpose, to avoid distracting beginners with details). The most straightforward ways of spelling out the details lead to models which have to be abandoned in favor of the *environment model* in order to explain side-effects. The Substitution Model described in these notes extends to handle side-effects quite well. In fact, the environment model can be understood as a way to *implement* the present Substitution Model efficiently.

A sequence of zero or more definitions followed by an expression at the end is called a *body*.

The only place Scheme grammar allows definitions to appear is at the beginning of bodies. Moreover, bodies which begin with definitions can only appear by themselves at "top-level," or as the bodies of lambda expressions (cf. Appendix A).

The lambda application rule deals with this by directing that the newly created definitions be inserted someplace that is grammatical. Definitions at the beginning of a lambda body may also have to be moved after the lambda disappears. For example, using the rule expressed above,

```
(if (zero? ((lambda (n) (define m 2) (+ m n)) 3)) 4 5)
```

would rewrite to

```
(if (zero? (define n 3) (define m 2) (+ m n)) 4 5)
```

which is not grammatical because the operand of `zero?` is a body with defines. Proper use of the lambda application rule instead rewrites this conditional as a body:

```
(define n 3) (define m 2) (if (zero? (+ m n)) 4 5)
```

2.2.2 Fresh Renaming

With lambda application, however, there is a naming problem we cannot avoid. For example, in the definitions below, `add2` is defined as a procedure that adds two to its operand; similarly, `add3` adds three.

```
(define n 2)
(define add2 (lambda (m) (+ m n)))
(define add3 (lambda (n) (+ 1 (add2 n))))
(add3 5)
```

So in Scheme, evaluation of the following body returns the value 8, as we would expect. Let's examine the steps in this computation. We begin by instantiating the underlined occurrence of `add3`:

```
(define n 2)
(define add2 (lambda (m) (+ m n)))
(define add3 (lambda (n) (+ 1 (add2 n))))
((lambda (n) (+ 1 (add2 n))) 5)
```

It will be easier to follow this example if we now "garbage collect"—that is, erase—the underlined definition of `add3` since it is no longer referenced. This leaves:

```
(define n 2)
(define add2 (lambda (m) (+ m n)))
((lambda (n) (+ 1 (add2 n))) 5)
```

Now the only possible rewriting is lambda application, yielding:

```
(define n 2)
(define add2 (lambda (m) (+ m n)))
(define n 5)
(+ 1 (add2 n))
```

Here we have an obvious problem: two definitions of `n` at the beginning of the block. This is ambiguous, and indeed violates the Scheme grammatical rule that a sequence of definitions may contain only one definition per variable. We handle this by renaming the newly defined `n` and the occurrences it binds, to some "fresh" name, say `n_1`:

```
(define add2 (lambda (m) (+ m n)))
(define n 2)
(define n_1 5)
(+ 1 (add2 n_1))
```

This expression is grammatical again, and now it is safe to instantiate and garbage-collect `add2`:

```
(define n 2)
(define n_1 5)
(+ 1 ((lambda (m) (+ m n)) n_1))
```

Notice that if we had not renamed `n` to `n_1` in this way, we would have rewritten to

```
(define n 2)
(define n 5)
(+ 1 ((lambda (m) (+ m n)) n))
```

and the ambiguity of which occurrence of `n` is defined to be which value becomes unresolvable.[7]

So the final, general formulation of the *lambda application rule* is:

$$\texttt{((lambda (}\langle\text{variable}_1\rangle \ \ldots\ \texttt{)}\ \langle\text{define}\rangle^*\ \langle\text{expression}\rangle\texttt{)}\ \langle\text{immediate value}_1\rangle\ \ldots\ \texttt{)}$$
$$\longrightarrow \langle\text{expression}\rangle$$

[7]Actually, until the mid-1970's, all Lisp dialects resolved the ambiguity by choosing the "nearest" value of `n`, so that this example would have rewritten to `(+ 1 ((lambda (m) (+ m 5)) 5))` and resulted in the value 11 instead of 8. This method of name management is called "dynamic scope."

with the sequence

$$\texttt{(define } \langle\text{variable}_1\rangle\ \langle\text{immediate value}_1\rangle\texttt{)} \ldots \langle\text{define}\rangle^*$$

placed above ⟨expression⟩ in the first position where definitions may appear and *with all these defined variables freshly renamed.*

Here is an example illustrating this general form of the rule.

Example.

```
(define f (lambda (x) (+ x 2)))
(if (zero? ((lambda (g x) (define f (lambda (x) (* x x))) (f (g x)))
            (lambda (x) (f (- x 2)))
            -1)
    3
    4)
    ⟶ (define f (lambda (x) (+ x 2)))
       (define g_0 (lambda (x) (f (- x 2))))
       (define x_0 -1)
       (define f_1 (lambda (x) (* x x)))
       (if (zero? (f_1 (g_0 x_0)))
           3
           4)
```

A more detailed discussion of fresh renaming appears in Appendix B.

2.3 Garbage Collection

"Garbage collection" refers to the process whereby Lisp-like systems identify and recapture previously used, but no longer accessible, storage cells. The corresponding process in our substitution model is erasure of unreferenced definitions, as in the previous example.

So, a subset G of the immediate definitions occurring at the beginning of a body is said to be *garbage in the body*, if none of the variables left in the body after G is erased were defined by G. The Substitution Model *garbage collection rule* is: *erase garbage.*

For example, in

```
(define a (lambda () b))
(define b 3)
(define c (lambda () (* b d)))
(define d 4)
(+ 1 (a))
```

The definitions of `c` and `d` are *garbage* because `c`'s and `d`'s don't occur anywhere outside their own definitions. The body can be garbage collected to be:

```
(define a (lambda () b))
(define b 3)
(+ 1 (a))
```

More details of garbage collection are given in Appendix C.

3 Uniqueness

A substitution model in which rules can be used anywhere is called a *Term Rewriting Model* (TRM). That's essentially what we have described above for functional Scheme. We need not specify where or in what order to apply rewrite rules because even the full set of rules shares the unique value property of simple algebraic rules.

Definition 1. We use the notation $B_1 \longrightarrow^* B_2$ to indicate that body B_1 can be rewritten to body B_2 by successively applying the Scheme rewriting rules above zero or more times.

> **The Unique Primitive Value Theorem**. *If Scheme body B_1 can be rewritten to some* primitive *value, p, then p is the only immediate value of any kind to which it can rewritten.*

In other words, if $B_1 \longrightarrow^* p$ and also $B_1 \longrightarrow^* V$ for some $\langle$immediate value$\rangle$, V, then $V = p$.

That's the good news. The bad news is that the order in which rules are applied *does* matter to avoid wasted time and space—even infinite waste if rules are applied forever in useless places. For example, one could waste a lot of time uselessly rewriting the subexpression `(Fibonacci 1000)` in:

```
(if #t 1 (Fibonacci 1000))
```

More bad news is that an expression may rewrite to many distinct immediate values none of which are terminated, for example,

```
    (lambda (x) (define y (lambda (z) (+ x (y z)))) (y 1))
→* (lambda (x) (define y (lambda (z) (+ x (y z)))) (+ x (y 1)))
→* (lambda (x) (define y (lambda (z) (+ x (y z)))) (+ x (+ x (y 1))))
      ⋮
```

This last example also illustrates that the full set of TRM rules do not have the Termination property.

These non-uniqueness problems can largely be avoided by keeping to the Lambda Body Rewriting Restriction mentioned above. In the functional case, this restriction never prevents rewriting to some value:

> **Lambda Body Safety Theorem**. *If a body can be rewritten into a value by successive use of the functional Scheme rewrite rules above, then it can be rewritten to a value while obeying the Lambda Body Rewriting Restriction.*

Of course by the Uniqueness Theorem, rewriting according to the Lambda Body Restriction will reach the same *primitive* value, if any, as any other way of applying the rules.

The Uniqueness and Safety Theorems are by no means obvious. (Their proof is sometimes covered in the graduate course 6.840 on Semantics of Programming Languages.) Term Rewriting Theory is a sub-discipline of Theoretical Computer Science which focuses on such properties.

4 Rule-specified Procedures

Since the built-in Scheme procedure `not` may be applied to any value, not just primitive values, we have to go beyond primitive application rules to describe its behavior. So we provide specific rules for applying `not`:

`(not #f)` $\longrightarrow$ `#t`,
`(not` V`)` $\longrightarrow$ `#f`,
when V is an $\langle$immediate value$\rangle$ not equal to `#f`.

Notice that in order to use `not` as an operator, the rules require that its operand be an $\langle$immediate value$\rangle$. This is the essential property of procedures in a call-by-value language like Scheme, and so we classify `not` as a *rule-specified* procedure identifier.

Another such rule-specified procedure identifier is, for example, `integer?` with rules:

`(integer?` V`)` $\longrightarrow$ `#t`
when V is the numeral for an integer,
`(integer?` V`)` $\longrightarrow$ `#f`
when V is an $\langle$immediate value$\rangle$ other than a numeral for an integer.

Rule-specified procedure identifiers are, like primitive procedure identifiers, also counted as immediate values.

5 Derived Expressions

We have described above a "kernel" of Scheme which omits many familiar, convenient Scheme constructs. For example, we want to handle `cond` and `let`. One approach is to add rewrite rules to handle these special forms directly. Another approach is to think of these extensions as abbreviations, or "syntactic sugar," for kernel expressions. For example,

$$\texttt{(let ((}\langle\text{variable}_1\rangle\ \langle\text{init}_1\rangle\texttt{)...)}\ \langle\text{body}\rangle\texttt{)}$$

can be understood as an abbreviation for

$$\texttt{((lambda (}\langle\text{variable}_1\rangle\cdots\texttt{)}\ \langle\text{body}\rangle\texttt{)}\ \langle\text{init}_1\rangle\ \texttt{...).}$$

and

$$\begin{array}{l}
\texttt{(cond ((}\langle\text{test}_1\rangle\ \langle\text{expression}_1\rangle\texttt{)} \\
\qquad\quad \texttt{((}\langle\text{test}_2\rangle\ \langle\text{expression}_2\rangle\texttt{)} \\
\qquad\qquad\qquad \vdots \\
\qquad\quad \texttt{(else}\ \langle\text{expression}_n\rangle\texttt{))}
\end{array}$$

can be understood as an abbreviation for

$$\begin{array}{l}
\texttt{(if}\ \langle\text{test}_1\rangle \\
\quad \langle\text{expression}_1\rangle \\
\quad \texttt{(if}\ \langle\text{test}_2\rangle \\
\qquad \langle\text{expression}_2\rangle \\
\qquad \vdots \\
\qquad\quad \texttt{(if}\ \langle\text{test}_{n-1}\rangle \\
\qquad\qquad \langle\text{expression}_{n-1}\rangle \\
\qquad\qquad \langle\text{expression}_n\rangle\texttt{)}\ \ \texttt{...}\ \ \texttt{)).}
\end{array}$$

So we can specify the computations of ⟨derived expression⟩'s by translating, or "desugaring," them into kernel ⟨expression⟩'s.[8] The Revised[4] Scheme Report describes the translations above and others for ⟨and⟩'s, ⟨or⟩'s, and several further forms.

Problem 3. The procedure `not` could have been treated as sugar instead of as a rule-defined procedure. Describe a simple translation of `not` into kernel Scheme.

[8]These translations are generally linear-time, one-pass, and yield a kernel-⟨expression⟩ of size proportional to the original extended-⟨expression⟩. Real Scheme interpreters and compilers typically carry out such "source to source" translations.

6 Lists and Symbols

Scheme output notation for lists and symbols makes them look just like expressions. Using this notation in our Term Rewriting Model presents a problem: how do we tell the difference between lists and combinations?[9] For example, is "`(+ 3 4)`" a combination—which should be rewritten to the syntactic value `7`—or is it a list of three elements—the procedure identifier `+` (or perhaps it is the symbol "`+`") followed by `3` and `4`? To avoid any confusion, the TRM instead rewrites lists into nested `cons`'s of list elements. For example, the expression `(list + 3 4)`, which Scheme prints out as "`([compiled arithmetic procedure +] 3 4)`," would be rewritten in our TRM into the expression

`(cons + (cons 3 (cons 4 ())))`.

Similarly, *symbols* which print out as `a` or `+` would be described in the TRM by the expressions `(quote a)` or `(quote +)` to avoid confusing them with variables or procedure identifiers. So `(list '+ 3 4)`, which Scheme prints out as "`(+ 3 4)`," would would be rewritten in our TRM into the terminated expression

`(cons (quote +) (cons 3 (cons 4 ())))`.

With these notational conventions, it is a straightforward to handle list structures in the TRM[10]. Namely, we specify that *symbol expressions* of the form

`(quote` ⟨identifier⟩`)`

and the *nil expression* `()` are also primitive values, we introduce new rule-specified procedure identifiers for list operations such as `cons`, `car` and `cdr`, and we extend the definitions of ⟨immediate value⟩ to include all expressions of the form

`(cons` ⟨immediate value⟩ ⟨immediate value⟩`)`.

Now it turns out that the behavior of `cons` is specified by having no rewrite rule for it at all! We do add some obvious rules for `car` and `cdr`:

$$\texttt{(car (cons } V_1\ V_2\texttt{))} \longrightarrow V_1$$
$$\texttt{(cdr (cons } V_1\ V_2\texttt{))} \longrightarrow V_2$$

[9] The fact that lists print out looking like expressions is generally considered a valuable feature in Lisp. On the other hand, at least one famous contemporary Computer Scientist has protested publicly that because of this he couldn't learn Lisp—he could never figure out when to stop evaluating expressions.

[10] This way of handling list works usefully until we begin mutating lists with `set-car!` and `set-cdr!`. Then we will need a notation which more accurately reflects sharing in list structures.

Here are the remaining important rewriting rules about lists:

$$\texttt{(apply } V_0 \texttt{ (cons } V_1 \ldots \texttt{ (cons } V_n \texttt{ ()) ...))} \longrightarrow \texttt{(}V_0\ V_1 \ldots V_n\texttt{)}$$

$$\texttt{(pair? (cons } V_1\ V_2\texttt{))} \longrightarrow \texttt{\#t}$$
$$\texttt{(pair? } V_3\texttt{)} \longrightarrow \texttt{\#f},$$
$$\text{if } V_3 \text{ is not of the form } \texttt{(cons } V_1\ V_2\texttt{)}.$$

$$\texttt{(null? ())} \longrightarrow \texttt{\#t}$$
$$\texttt{(null? } V\texttt{)} \longrightarrow \texttt{\#f},$$
$$\text{if } V \text{ is not } \texttt{()}.$$

$$\texttt{(eq? (quote } I\texttt{) (quote } I\texttt{))} \longrightarrow \texttt{\#t}$$
$$\texttt{(eq? (quote } I\texttt{) (quote } J\texttt{))} \longrightarrow \texttt{\#f},$$
$$\text{if } I \text{ and } J \text{are distinct } \langle\text{identifier}\rangle\text{'s.}$$

$$\texttt{(list)} \longrightarrow \texttt{()}$$
$$\texttt{(list } V_1 \texttt{ ...)} \longrightarrow \texttt{(cons } V_1 \texttt{ (list ...))}$$

7 Running the Substitution Model

There is an implementation of the Substitution Model available through the 6.044 WWW page. It may help you understand how Scheme processes behave, and it can also serve as a simple debugging aid. Of course it is about four orders of magnitude slower than Scheme, so don't expect it to be useful on compute-intensive examples.

The order in which rewrite rules are applied in the implementation reflects Scheme evaluation order fairly accurately. To generate the list of expressions to which an input expression rewrites, apply the procedure `smeval` to the quoted expression.

For example, evaluate

```
(smeval '((lambda (n) (+ 2 n)) 3))
```

to see the successive expressions as this lambda application rewrites to `5`.

A body beginning with one or more definitions is represented as a list of the definitions with an expression at the end. So evaluating,

```
(smeval
 '((define (factorial n)
     (if (= n 0)
```

```
            1
            (* n (factorial (- n 1)))))
      (factorial 4)))
```

will allow you to watch how this body rewrote to `24`.

An expression printed out by `smeval` at any point is a well-formed Scheme expression which can be evaluated in Scheme and will give the same final result as `smeval`—at least if the final result is a number or truth value[11].

Our TRM does not model Scheme's ability to remember definitions from one evaluation to the next. For example, evaluating

```
(define (square n) (* n n))
```

in Scheme will return an unspecified value, after which evaluating

```
(square 3)
```

will return `9`. But if you try

```
(smeval '(define (square n) (* n n)))
```

you will get some (probably uninformative) error message. The reason for the error is that `smeval` expects a *body* as an argument, and a definition by itself is not a body. Instead, you should have typed

```
(smeval
 '((define (square n) (* n n))
   (square 3)))
```

7.1 Controlling Printout

There are several utilities to control the output printed by `smeval`.

The value returned by a call to `smeval` is the ⟨value⟩ at which it successfully stops, or else an error message.

[11]To evaluate a *body* printed by `smeval` when the body begins with definitions, it has to be made into a Scheme expression. If the body printed out is

$$(\langle \text{define}_1 \rangle \ldots \langle \text{expression} \rangle),$$

then in Scheme evaluate

$$(\texttt{(lambda ()}\ \langle \text{define}_1 \rangle \ldots \langle \text{expression} \rangle\texttt{))}.$$

The procedure `print-this-step?` determines which steps get saved on the list generated by `smstep-list` for printing by `smeval`. The default is to print every step.

The procedure `garbage-collect-this-step?` can be used to control the frequency of garbage collection. Its default is to garbage-collect at every step (which can slow things down a bit). To postpone all garbage collection until the last step, evaluate

```
(define (garbage-collect-this-step? step-number body) #f)
```

The procedure `interrupt-smeval?` limits the number of rewriting steps in an evaluation. Its default is to abort evaluation after 600 steps. You may want to do shorter runs. For example, to set `smeval` to stop after 100 rewriting steps, evaluate

```
(define (interrupt-smeval? step-number body) (> step-number 100))
```

Warnings: The code for the TRM implementation is written to be readable by (ambitious) beginning 6.001 students, with major sacrifices in speed and resilience for the sake of clarity. As a consequence, `smeval` typically runs forever or crashes gracelessly on ungrammatical inputs. It is a good idea to test expressions by evaluating them in Scheme before `smeval`'ing them.

Edwin itself wedges horribly if the `*scheme*` buffer overflows (which may happen when its size is within a small factor of 100K characters). You don't want this to happen! So be careful about doing `smeval` for long computations; instead save the list generated by procedure `smstep-list` and view it selectively.

A Grammar for Functional Scheme

In the following grammars, superscript "$*$" indicates zero or more occurrences of a grammatical phrase.

A.1 Kernel Syntax

$$
\begin{aligned}
\langle\text{define}\rangle &:= \langle\text{immediate define}\rangle \\
\langle\text{immediate define}\rangle &:= \texttt{(define } \langle\text{variable}\rangle\ \langle\text{immediate value}\rangle\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{body}\rangle &::= \langle\text{define}\rangle^{*}\ \langle\text{expression}\rangle \\
&\qquad\quad \text{(Note: all defined variables must be distinct.)} \\
\langle\text{expression}\rangle &::= \langle\text{variable}\rangle \mid \langle\text{self-evaluating}\rangle \mid \langle\text{procedure identifier}\rangle \\
&\quad\ \mid \langle\text{symbol}\rangle \mid \texttt{()} \mid \langle\text{combination}\rangle \mid \langle\text{lambda expression}\rangle \mid \langle\text{if}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{self-evaluating}\rangle &::= \langle\text{numeral}\rangle \mid \langle\text{boolean}\rangle \\
\langle\text{numeral}\rangle &::= \texttt{0} \mid \texttt{-1} \mid \texttt{3.14159} \mid \ldots \\
\langle\text{boolean}\rangle &::= \texttt{\#t} \mid \texttt{\#f}
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{procedure identifier}\rangle &::= \langle\text{primitive procedure id}\rangle \mid \langle\text{rule-specified procedure id}\rangle \\
\langle\text{primitive procedure id}\rangle &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{=} \mid \texttt{<} \mid \texttt{atan} \mid \texttt{zero?} \mid \ldots \\
\langle\text{rule-specified procedure id}\rangle &::= \texttt{not} \mid \texttt{boolean?} \mid \texttt{integer?} \mid \texttt{number?} \mid \texttt{null?} \mid \texttt{pair?} \\
&\quad\ \mid \texttt{symbol?} \mid \texttt{procedure?} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{cons} \mid \texttt{list} \\
&\quad\ \mid \texttt{apply} \mid \texttt{eq?}
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{symbol}\rangle &::= \texttt{(quote}\ \langle\text{variable}\rangle\texttt{)} \mid \texttt{(quote}\ \langle\text{procedure identifier}\rangle\texttt{)} \\
&\quad\ \mid \texttt{(quote}\ \langle\text{keyword}\rangle\ \texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{combination}\rangle &::= \texttt{(}\ \langle\text{operator}\rangle\ \langle\text{operand}\rangle^*\ \texttt{)} \\
\langle\text{operator}\rangle &::= \langle\text{expression}\rangle \\
\langle\text{operand}\rangle &::= \langle\text{expression}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{lambda expression}\rangle &::= \texttt{(lambda (}\ \langle\text{formals}\rangle\ \texttt{)}\ \langle\text{body}\rangle\ \texttt{)} \\
\langle\text{formals}\rangle &::= \langle\text{variable}\rangle^* \qquad\qquad \text{(Note: all } \langle\text{variable}\rangle\text{'s must be distinct.)}
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{if}\rangle &:= \texttt{(if}\ \langle\text{test}\rangle\ \langle\text{consequent}\rangle\ \langle\text{alternative}\rangle\ \texttt{)} \\
\langle\text{test}\rangle &::= \langle\text{expression}\rangle \\
\langle\text{consequent}\rangle &::= \langle\text{expression}\rangle \\
\langle\text{alternative}\rangle &::= \langle\text{expression}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{keyword}\rangle &::= \texttt{define} \mid \texttt{quote} \mid \texttt{lambda} \mid \texttt{if} \\
\langle\text{variable}\rangle &::= \text{identifiers which are not } \langle\text{self-evaluating}\rangle\text{ ,} \\
&\qquad \langle\text{procedure identifier}\rangle\text{'s or } \langle\text{keyword}\rangle\text{'s}
\end{aligned}
$$

A.2 Values

$$
\begin{aligned}
\langle\text{primitive value}\rangle &::= \langle\text{self-evaluating}\rangle \mid \langle\text{symbol}\rangle \mid \texttt{()} \\
\langle\text{immediate value}\rangle &::= \langle\text{primitive value}\rangle \mid \langle\text{procedure identifier}\rangle \mid \langle\text{lambda expression}\rangle \\
&\quad\ \mid \texttt{(cons}\ \langle\text{immediate value}\rangle\ \langle\text{immediate value}\rangle\texttt{)} \\
\langle\text{value}\rangle &::= \langle\text{immediate define}\rangle^*\ \langle\text{immediate value}\rangle \\
&\qquad \text{(Note: all defined } \langle\text{variable}\rangle\text{'s must be distinct.)}
\end{aligned}
$$

A.3 Derived Syntax

⟨define⟩ ::= ... | `(define (`⟨variable⟩ ⟨formals⟩`)` ⟨body⟩`)`
⟨expression⟩ ::= ... | ⟨derived expression⟩
⟨derived expression⟩ ::= ⟨cond⟩ | ⟨let⟩

⟨cond⟩ ::= `(cond` ⟨clause⟩ ⟨clause⟩* `)`
⟨clause⟩ ::= `(` ⟨test⟩ ⟨expression⟩ `)` | `(else` ⟨expression⟩`)`

⟨let⟩ ::= `(let (`⟨binding⟩* `)` ⟨body⟩ `)`
(Note: all `let`-bound variables must be distinct.)
⟨binding⟩ ::= `(` ⟨variable⟩ ⟨init⟩ `)`
⟨init⟩ ::= ⟨expression⟩
⟨keyword⟩ ::= ... | `cond` | `let` | `else`

B Renaming Variables for Static Scope

The scope, or binding, rules of a language determine how names are associated with things named. Scheme is a *statically* scoped language (also called *lexically* scoped). The purpose of this scoping discipline is to ensure that formals or defined variables within an expression can be "kept private" and behave independently of occurrences of names outside the expression. In this way, a programmer can choose names within a program without concern for whether or how the same names may have been used elsewhere.

Correctly managing names is a problem only when the same variable is bound in more than one way in a body. In the example in Section 2.2.2, there were initially two places where `n` was bound: by the initial ⟨define⟩the definition of and as a formal in the definition of `add3`. There would have been no problem if these different bindings used different names in the first place, as in:

```
(define n 2)
(define add2 (lambda (m) (+ m n)))
(define add3 (lambda (n) (+ 1 (add2 n))))
(add3 5)
```

The lambda application rule specifies that such fresh renaming of `n` to `n_1` should be done as definitions are created and/or moved to the beginning of a block. To understand fresh renaming precisely, we must understand exactly how to determine which occurrences of variables are bound where. So we will now carefully spell out the scoping rules for Scheme.

An occurrence of a variable in a body is a *binding occurrence* if it occurs as a formal of a lambda expression or as the variable part of a definition. Associated with every binding occurrence is a subbody called the *scope* of the binding.

- The scope of a formal of a lambda expression is the body of the lambda expression.
- The scope of the variable part of a given definition is the subbody consisting of the *entire sequence* of definitions in which the given definition appears, as well as the expression at the end of the sequence.

For example, consider the following body beginning with two definitions:

```
(define odd?                                            ;works on any numbers
  (lambda (n)
   (if (integer? n)
       (= (parity n) 1)
       false)))
(define parity
  (lambda (n)
    (define even?
             (lambda (n)
               (cond ((zero? n) #t)
                     ((< n 0) (odd? (+ 1 n)))
                     ((> n 0) (odd? (- n 1))))))
    (define odd?                                       ;only works on integers
             (lambda (n)
               (cond ((zero? n) false)
                     ((< n 0) (even? (+ 1 n)))
                     ((> n 0) (even? (- n 1))))))
    (if (even? n) 1 0)))
(odd? (+ n 0.3))
```

In this example, there is only one binding occurrence of `parity`, namely as the variable in the second definition in the body. The scope of this binding occurrence is the *whole* body. There are two binding occurrences of `odd?`. The first is as the variable in the second definition in the subbody:

```
(define even?
         (lambda (n)
           (cond ((zero? n) #t)
                 ((< n 0) (odd? (+ 1 n)))
                 ((> n 0) (odd? (- n 1))))))
(define odd?
         (lambda (n)
```

```
            (cond ((zero? n) false)
                  ((< n 0) (even? (+ 1 n)))
                  ((> n 0) (even? (- n 1))))))
(if (even? n) 1 0)
```

and its scope is this whole subbody. The second binding occurrence of `odd?` is as the variable in the definition at the beginning of the whole body, and its scope is the whole body. There are four binding occurrences of `n` as the formal of four different lambda expressions within the body.

Any occurrence of a variable which is not a binding occurrence and which is inside the scope of a binding occurrence of that variable, is called a *bound occurrence* of that variable. The bound occurrence is said to be *bound to* the binding occurrence with the smallest scope which includes it. We can illustrate which bound occurrences are bound to which binding occurrences by attaching the same numerical suffix to binding occurrences and occurrences bound to them:

```
(define odd_1?
  (lambda (n_3)
   (if (integer? n_3)
       (= (parity_0 n_3) 1)
       false)))
(define parity_0
  (lambda (n_2)
    (define even?_0
             (lambda (n_0)
               (cond ((zero? n_0) #t)
                     ((< n_0 0) (odd?_0 (+ 1 n_0)))
                     ((> n_0 0) (odd?_0 (- n 1_0))))))
    (define odd?_0
             (lambda (n_1)
               (cond ((zero? n_1) false)
                     ((< n_1 0) (even?_0 (+ 1 n_1)))
                     ((> n_1 0) (even?_0 (- n 1_1))))))
    (if (even?_0 n_2) 1 0)))
(odd_1? (+ n 0.3))
```

Notice that the first occurrence of the variable `parity_0` actually occurs *before* the definition which binds it!

The occurrences in a body which are neither binding nor bound are called *free*. Variables with free occurrences are called the free variables of the body. In the example above, the free variables are the ones left without any number suffix: namely `false` and `n`. (Note that `integer?` and `zero?` don't count as free variables because they are procedure identifiers.)

There is a more procedural way to define the *free variables in a body*. If the body is

- ⟨primitive value⟩ or ⟨procedure identifier⟩: the body has no free variables.
- a variable x: the body has x as its only free variable.
- a conditional: the free variables are the variables which are free in either the test, consequent, or alternate expressions.
- a combination: the free variables are the variables which are free in operator or operands.
- a lambda expression: the free variables are the free variables of the body of the lambda expression *excluding* the variables which are formals of the lambda expression.
- a body beginning with one or more definitions: the free variables are the free variables in the immediate value parts of the definitions at the beginning, plus the free variables of the expression at the end of the body, *excluding* the variables defined at the beginning of the body.

C A Garbage Collection Procedure

The definition of "garbage in the body" we gave in Section 2.3 was accurate, but not very procedural. An efficient way to find all the garbage in a body is to identify all the variables which are needed in the body—the rest are garbage. Here is an inductive definition of the *variables needed in a body* which effectively specifies a procedure for finding all the garbage.

- All free variables of the expression at the end of the body are needed in the body.
- If x is a variable needed in a body and `(define` x V`)` is one of the defines at the beginning of the body, then all the free variables of V are also needed in the body.