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,

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

In general, a primitive application rule is of the form

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

where ⟨primitive value⟩'s are objects which have standard printable output representations—like numbers or truth values—-and primitive procedure variables are identifiers like `+`, `<`, and `atan` with "built-in" meaning in Scheme.

By repeatedly *applying* such rules, namely replacing lefthand sides of rules with their righthand 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 *truth values* `#t`, `#f`:

$$\texttt{(< 6 7)} \longrightarrow \texttt{\#t}$$
$$\texttt{(not \#f)} \longrightarrow \texttt{\#t}$$
$$\texttt{(zero? -1.414)} \longrightarrow \texttt{\#f}$$
$$\vdots$$

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

```
(not (< (+ 0 1 2) (/ 3 1))) ⟶ (not (< (+ 0 1 2) 3))
                            ⟶ (not (< 3 3))
                            ⟶ (not #f)
                            ⟶ #t
```

It is easy to see that *it does not matter in what order the primitive rewriting rules are applied*—the primitive value at the end will always be the same![1] For example, we might have chosen to rewrite the example above in another order:

```
(not (< (+ 0 1 2) (/ 3 1))) ⟶ (not (< 3 (/ 3 1)))
                            ⟶ (not (< 3 3))
                            ⟶ (not #f)
                            ⟶ #t
```

The primitive value is unique because Scheme is *fully parenthesized*—ambiguous mathematical expressions like $(3 + 2 - 4)$ are disallowed—and because there are no primitive rules with the same lefthand side but different righthand sides.

1.2 Immediate Values

In addition to the primitive values, the primitive *operators* like `+` or `<` are themselves a kind of value called a *primitive procedure variable*[2].

Scheme also provides a means to construct procedures values, namely, with *lambda expressions*. The

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

are called *immediate values*.

[1] Well, there's a *proviso* here: the primitive value is unique when it is *possible* somehow to rewrite the expression into one. Some "ill-typed" expressions like `(+ #t 3)` do not have values. (What would Scheme do when such an expression is evaluated?)

[2] We assume here that primitive procedure variables *won't* be redefined! According to the official definition of Revised^4 Scheme, it is theoretically 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 simplification of combinations involving an arbitrary expression, A, which may contain variables, as opposed merely to 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:

$$
\begin{aligned}
\texttt{(if \#f } \langle\text{expression}_1\rangle\langle\text{expression}_2\rangle\texttt{)} &\longrightarrow \langle\text{expression}_2\rangle \\
\texttt{(if } \langle\text{non-}\texttt{\#f}\text{ immediate value}\rangle\,\langle\text{expression}_1\rangle\langle\text{expression}_2\rangle\texttt{)} &\longrightarrow \langle\text{expression}_1\rangle
\end{aligned}
$$

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. Thus, in contrast to primitive applications, the `if` rule can be applied *without* the constraint that $\langle\text{expression}_1\rangle$ and $\langle\text{expression}_2\rangle$ be values.

Now we can do rewriting such as

```
(if (lambda (a b) c) (if (not (< (+ 0 1 2) (/ 3 1))) atan (lambda (x) x)) +)
                  ⟶ (if (not (< (+ 0 1 2) (/ 3 1))) atan (lambda (x) x))
                        ⋮
                  ⟶ (if #t atan (lambda (x) x))
                  ⟶ atan
```

As long as we restrict to a *functional* subset of Scheme[3], i.e., Scheme without side-effects, it will still be the case that no matter the order that primitive and conditional rewriting rules are applied, the same immediate value, if any, will result. 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.

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

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.

2 Substitution Models

Another familiar part of algebra is definition of names and their use, as in,

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

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

The variable instantiation is a basic naming rule of algebra: replace an occurrence of a variable by the expression it names. For example, using instantiation and primitive applications in Scheme, we can carry out the following calculation:

```
(define y 7) (define x (+ 1 2)) (* x y)
    ⟶ (define y 7) (define x (+ 1 2)) (* x 7)
    ⟶ (define y 7) (define x 3) (* x 7)
    ⟶ (define y 7) (define x 3) (* 3 7)
    ⟶ (define y 7) (define x 3) 21
```

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.

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].

Second, variables may only be instantiated with *immediate values*, not arbitrary expressions. This restriction is needed to reflect accurately Scheme's call-by-value behavior and sharing among environment frames. For example, unlike instantiation in algebra, Scheme's instantiation rule does *not* allow

```
(define y 7) (define x (+ y y)) (* x 3 y )
    —→ (define y 7) (define x (+ y y)) (* (+ y y) 3 y)
```

because the variable `x` may not be instantiated by the non-⟨immediate value⟩ `(+ y y)`.

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) (inc (add2 n))))
```

so that evaluating `(add3 1)` would return `4`.

But suppose we naively instantiate the second occurrence of `add2`:

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

Now, evaluating `(add3 1)` would return `3`!

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 for changing defined names into "fresh" names. However, there is no need for renaming, and no problem about changed binding, as long as we obey

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

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

This works because in Scheme grammar, the only place definitions can occur besides "top level" is within lambda bodies, so there is no possibility of bindings being changed by instantiating ocurrrences 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 instantiation (without renaming) inside some lambda body in M.

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 expressions to be manipulated in many examples. It also will allow us to model Scheme's computational behavior much more accurately.[5]

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 a new "garbage collection" rule which allows us to remove unreferenced definitions.

[5]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 usefully be understood as a way to *implement* the present Substitution Model efficiently.

2.2.1 Bodies

Note that we might have tried expressing the rule above as

```
((lambda (x) B) V)
        ⟶ (define x V) B.
```

This version introduces a slight grammatical complication: the expression on the lefthand side of the rule rewrites into a sequence of two things—a definition followed by an expression. If there were several formal parameters instead of just `x`, there would be several definitions preceding the expression.

> 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 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 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. So in Scheme, evaluation of the following body returns the value 8, as we would expect:

```
(define make-incrementer (lambda (n) (lambda (m) (+ m n))))
(define add2 (make-incrementer 2))
(define add3 (lambda (n) (inc (add2 n))))
(add3 5)
```

Let's examine in detail how our Substitution Model realizes this intended behavior. The only rule applicable at the start is instantiatiation for `make-incrementer` or `add3`. Scheme would work on the definitions at the beginning first, so let's do likewise and first instantiate the occurrrence of `make-incrementer` underlined above:

```
(define make-incrementer (lambda (n) (lambda (m) (+ m n))))
(define add2 ((lambda (n) (lambda (m) (+ m n))) 2))
(define add3 (lambda (n) (inc (add2 n))))
(add3 5)
```

It will be easier to follow this example if we now simplify the body by dropping the no longer referenced definition—in this case, the underlined definition of `make-incrementer`:

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

Lambda application in the definition of `add2` creates a definition of `n`:

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

Now we can instantiate `add3` and then drop its definition:

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

Lambda application comes next, yielding:

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

and the ambiguity of which occurrence of `n` is defined to be which value becomes unresolvable.[6]

So the final, accurate statement of the *lambda application rule* is:

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

with the sequence

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

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

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 dropping of unreferenced definitions as in the previous example. A subset G of immediate definitions among the definitions occurring at the beginning of a body

[6] 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 `(inc ((lambda (m) (+ m 5)) 5))` and resulted in the value 11 instead of 8. This method of name management is called "dynamic scope."

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:

$$\langle \text{body}_1 \rangle \longrightarrow \langle \text{body}_2 \rangle,$$
$$\text{where } \langle \text{body}_2 \rangle \text{ is the result of erasing garbage in } \langle \text{body}_1 \rangle.$$

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 what we have described above for functional Scheme, because even the full set of rules shares the unique value property of simple algebraic rules:

> **The Unique Value Theorem**. *There is at most one primitive value that can be reached by successively rewriting a body applying the Scheme rewriting rules above to any subbodies in any order.*

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 `(fibonaaci 1000)` in:

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

More bad news is that immediate values which are *not* primitive, namely, lambda expressions, are rarely unique.

These problems can largely be avoided by keeping to the *lambda body rewriting restriction* mentioned above. There is no problem about this because of the

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

Of course by the Uniqueness Theorem, rewriting under the lambda body rewriting 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 subdiscipline of Theoretical Computer Science which focuses on such properties.

4 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,

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

can be understood as an abbreviation for

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

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

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

So we can specify the computations of ⟨derived expression⟩'s by translating, or "desugaring," them into kernel ⟨expression⟩'s[7]. The Revised[4] Scheme Report describes the translations above and others for ⟨and⟩'s, ⟨or⟩'s, and several further forms.

5 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[8]? 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 variable `+` (or perhaps it is the symbol "`+`") followed by `3` and `4`? So the TRM instead "simplifies" lists into syntactic values which are 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. So `(list '+ 3 4)`, which Scheme prints out as "`(+ 3 4)`," would would be rewritten in our TRM into the expression

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

With these notational conventions, it is a straightforward to handle list structures in the TRM[9]. Namely, we specify that *symbol expressions* of the form

`(quote` ⟨identifier⟩`)`

and the *nil expression* `()` are also primitive values, we introduce new list-oriented procedure variables 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⟩`).`

We don't even have to add a rewrite rule for applications of `cons`! We do add some obvious rules for `car` and `cdr`:

$$\texttt{(car (cons } \langle\text{immediate value}_1\rangle\ \langle\text{immediate value}_2\rangle\texttt{))} \longrightarrow \langle\text{immediate value}_1\rangle$$
$$\texttt{(cdr (cons } \langle\text{immediate value}_1\rangle\ \langle\text{immediate value}_2\rangle\texttt{))} \longrightarrow \langle\text{immediate value}_2\rangle$$

---

[7] 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 translations.

[8] The fact that lists print out looking like expressions is generally considered a very 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.

[9] This way of handling list works usefully until we begin mutating lists with `set-car!` and `set-cdr!`, at which point a more accurate notation reflecting sharing in list structure would be needed.

The rules for `cons`, `car` and `cdr` resemble the primitive procedure rules in that they require that all the operands be ⟨immediate value⟩'s before the rule can be implied. These *rule-specified procedure variables* are, like primitive procedure variables, also treated as ⟨immediate value⟩'s.

Here are the remaining important rewriting rules about lists:

$$\texttt{(apply } E_0 \texttt{ (cons } E_1 \ldots \texttt{ (cons } E_n \texttt{ ()) ...))} \longrightarrow \texttt{(}E_0\ E_1 \ldots E_n\texttt{)}$$

$$\texttt{(pair? (cons } \langle\text{immediate value}_1\rangle\ \langle\text{immediate value}_2\rangle\texttt{))} \longrightarrow \texttt{\#t}$$
$$\texttt{(pair? } \langle\text{immediate value}_3\rangle\texttt{)} \longrightarrow \texttt{\#f}$$

if ⟨immediate value$_3$⟩ is not of the form `(cons` ⟨immediate value$_1$⟩ ⟨immediate value$_2$⟩`)`

$$\texttt{(null? ())} \longrightarrow \texttt{\#t}$$
$$\texttt{(null? } \langle\text{immediate value}\rangle\texttt{)} \longrightarrow \texttt{\#f}$$

if ⟨immediate value⟩ is not `()`.

$$\texttt{(list)} \longrightarrow \texttt{()}$$
$$\texttt{(list } \langle\text{immediate value}_1\rangle \texttt{ ...)} \longrightarrow \texttt{(cons } \langle\text{immediate value}_1\rangle \texttt{ (list ...))}$$

$$\texttt{(equal? } \langle\text{immediate value}_1\rangle\ \langle\text{immediate value}_1\rangle\texttt{)} \longrightarrow \texttt{\#t}$$
$$\texttt{(equal? } \langle\text{immediate value}_1\rangle\ \langle\text{immediate value}_2\rangle\texttt{)} \longrightarrow \texttt{\#f}$$

if ⟨immediate value$_1$⟩ and ⟨immediate value$_2$⟩ are not identical,
and at least one of them is printable.

The *printable* values are numbers, Booleans, symbols, and lists of printable values. Note that, the expression

```
(equal? (lambda () 2) (lambda () (+ 1 1))
```

for example, could be rewritten to both `#t` and `#f` if we omit the printable value restriction on the rules for `equal?`.

6 Running the Substitution Model

There is an implementation of the Substitution Model in the 6.044 ftp directory. 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 (dec n)))))
   (factorial 4)))
```

will allow you to watch how this body rewrote to `24` (see Appendix D).

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[10].

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)))
```

[10]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

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

6.1 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.

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.

6.2 Bugs

We don't know of any now, but there likely are some. Gratitude and lots of 6.044 brownie points for bug reports—email them to `6044-lecturer@theory.lcs.mit.edu`.

A Grammar for Functional Scheme

A.1 Kernel Syntax

$$
\begin{array}{rcl}
\langle\text{define}\rangle & := & \texttt{(define}\ \langle\text{variable}\rangle\,\langle\text{expression}\rangle\,\texttt{)} \\
\langle\text{body}\rangle & ::= & \langle\text{define}_1\rangle \ldots \langle\text{expression}\rangle \\
 & & \qquad\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{symbol}\rangle \mid \texttt{()} \\
 & & \mid \langle\text{combination}\rangle \mid \langle\text{lambda expression}\rangle \mid \langle\text{if}\rangle \\
 & & \\
\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} \\
 & & \\
\langle\text{symbol}\rangle & ::= & \texttt{(quote}\ \langle\text{variable}\rangle\,\texttt{)} \mid \texttt{(quote}\ \langle\text{keyword}\rangle\texttt{)} \\
 & & \\
\langle\text{combination}\rangle & ::= & \texttt{(}\,\langle\text{operator}\rangle\,\langle\text{operand}_1\rangle \ldots\texttt{)} \\
\langle\text{operator}\rangle & ::= & \langle\text{expression}\rangle \\
\langle\text{operand}\rangle & ::= & \langle\text{expression}\rangle \\
 & & \\
\langle\text{lambda expression}\rangle & ::= & \texttt{(lambda (}\langle\text{formals}\rangle\texttt{)}\ \ \langle\text{body}\rangle\,\texttt{)} \\
\langle\text{formals}\rangle & ::= & \langle\text{variable}_1\rangle \ldots \\
 & & \qquad\text{(Note: all }\langle\text{variable}\rangle\text{'s must be distinct)} \\
 & & \\
\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 \\
 & & \\
\langle\text{primitive procedure variable}\rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{=} \mid \texttt{<} \mid \texttt{atan} \mid \ldots \\
\langle\text{rule-specified procedure variable}\rangle & ::= & \texttt{null?} \mid \texttt{pair?} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{cons} \mid \texttt{list} \mid \texttt{equal?} \mid \texttt{apply} \\
\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{ or }\langle\text{keyword}\rangle\text{'s}
\end{array}
$$

A.2 Derived Syntax

$$
\begin{aligned}
\langle\text{expression}\rangle &::= \ldots \mid \langle\text{derived expression}\rangle \\
\langle\text{derived expression}\rangle &::= \langle\text{cond}\rangle \mid \langle\text{let}\rangle \mid \ldots \\
\langle\text{cond}\rangle &::= \texttt{(cond } \langle\text{clause}_1\rangle \ldots\texttt{)} \\
\langle\text{clause}\rangle &::= \texttt{(}\,\langle\text{test}\rangle\,\langle\text{expression}\rangle\,\texttt{)} \mid \texttt{(else } \langle\text{expression}\rangle\texttt{)} \\
\langle\text{let}\rangle &::= \texttt{(let (}\,\langle\text{binding}_1\rangle \ldots\texttt{)}\;\; \langle\text{body}\rangle\,\texttt{)} \\
\langle\text{binding}\rangle &::= \texttt{(}\,\langle\text{variable}\rangle\,\langle\text{init}\rangle\,\texttt{)} \\
\langle\text{init}\rangle &::= \langle\text{expression}\rangle \\
\langle\text{define}\rangle &::= \ldots \mid \texttt{(define (}\,\langle\text{variable}\rangle\,\langle\text{formals}\rangle\,\texttt{)}\,\langle\text{body}\rangle\,\texttt{)} \\
\langle\text{keyword}\rangle &::= \ldots \mid \texttt{cond} \mid \texttt{let} \mid \texttt{else} \mid \ldots
\end{aligned}
$$

A.3 Values

$$
\begin{aligned}
\langle\text{primitive value}\rangle &::= \langle\text{self-evaluating}\rangle \mid \langle\text{symbol}\rangle \mid \texttt{()} \\
\langle\text{immediate value}\rangle &::= \langle\text{primitive value}\rangle \mid \langle\text{lambda expression}\rangle \\
&\quad \mid \langle\text{primitive procedure variable}\rangle \\
&\quad \mid \langle\text{rule-specified procedure variable}\rangle \\
&\quad \mid \texttt{(cons }\;\langle\text{immediate value}\rangle\;\langle\text{immediate value}\rangle\,\texttt{)} \\
\langle\text{immediate define}\rangle &::= \texttt{(define }\langle\text{variable}\rangle\,\langle\text{immediate value}\rangle\,\texttt{)}\ldots \\
\langle\text{value}\rangle &::= \langle\text{immediate define}_1\rangle \ldots \langle\text{immediate value}\rangle \\
&\qquad\qquad \text{(Note: all defined } \langle\text{variable}\rangle\text{'s must be distinct)}
\end{aligned}
$$

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: as a formal in the definition of `make-incrementor` 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 make-incrementor (lambda (n) (lambda (m) (+ m n))))
(define add2 (make-incrementor 2))
(define add3 (lambda (n_1) (inc (add2 n_1))))
(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 describe fresh renaming precisely, we must first 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)
       #f)))
(define parity
  (lambda (n)
    (define even?
            (lambda (n)
              (cond ((zero? n) #t)
                    ((< n 0) (odd? (inc n)))
                    ((> n 0) (odd? (dec n))))))
    (define odd?                                   ;only works on integers
            (lambda (n)
              (cond ((zero? n) #f)
                    ((< n 0) (even? (inc n)))
                    ((> n 0) (even? (dec n))))))
    (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? (inc n)))
                 ((> n 0) (odd? (dec n))))))
(define odd?
         (lambda (n)
           (cond ((zero? n) #f)
                 ((< n 0) (even? (inc n)))
                 ((> n 0) (even? (dec n))))))
(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)
       #f)))
(define parity_0
  (lambda (n_2)
    (define even?_0
             (lambda (n_0)
               (cond ((zero? n_0) #t)
                     ((< n_0 0) (odd?_0 (inc n_0)))
                     ((> n_0 0) (odd?_0 (dec n_0))))))
    (define odd?_0
             (lambda (n_1)
               (cond ((zero? n_1) #f)
                     ((< n_1 0) (even?_0 (inc n_1)))
```

```
                        ((> n_1 0) (even?_0 (dec n_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: the primitive procedure variables `integer?`, `zero?`, `inc`, and `dec`; and the ordinary variable `n`.

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

- number, truth value, symbol or `()`: the body has no free variables.
- variable x: the body has x as its only free variable.
- conditional (`if`): the free variables are the variables which are free in the test, consequent, or alternate expressions.
- combination: the free variables are the variables which are free in operator or operands.
- 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.
- body: the free variables are the free variables in the expression parts of the definitions at the beginning of the body, plus the free variables of the expression at the end of the body, *excluding* the variables defined at the beginning of the body.

C Garbage Collection in Detail

Here is a precise definition of garbage:

> A subset G of the ⟨define⟩'s at the beginning of a body is *garbage in the body* if
>
> 1. none of the variables defined by the ⟨define⟩'s in G occur free in the remaining body after G is erased, and
> 2. all the variables defined by the ⟨define⟩'s in G are bound to immediate values.

Condition 2 in the definition of garbage is a technicality needed to reflect call-by-value behavior. For example, in

```
(define spin (lambda () (spin)))
(define x (spin))
0
```

the definitions of `spin` and `x` are certainly not needed to find the value of the expression `0`, and indeed they satisfy condition 1 in the definition of garbage. But call-by-value means that the values of *all* the expressions in the definitions beginning a body must be obtained in order to find the final value of a body. Since `(spin)` has no value, neither does the body. Indeed, Scheme would run forever (if we didn't interrupt it) on this body. On the other hand, if we erased the definitions, we would finish with the primitive value 0. (That's what *call-by-name*, also called *normal order*, evaluation would return.)

The above definition of garbage is 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 a precise definition of "needed" which does suggest a procedure.

The variables needed in a body are defined inductively by the conditions:

1. all free variables in the expression at the end of the body are needed in the body,
2. a variable is needed in a body if it is bound by one of the definitions at the beginning of the body to an expression which is *not* an immediate value,
3. if x is variable needed in a body and `(define` x E`)` is one of the defines at the beginning of the body, then all the free variables of E are also needed in the body, and
4. variables are needed in the body only as a consequence of the preceding clauses 1–3.

D A Sample Run

```
(smeval
 '((define (factorial n)
     (if (= n 0)
         1
         (* n (factorial (dec n)))))
   (factorial 4)))

;==(0)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (factorial 4))

;==(1)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 ((lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))) 4))

;==(2)==>
```

```
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (if (= n_0 0) 1 (* n_0 (factorial (dec n_0)))))

;==(3)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (if (= 4 0) 1 (* n_0 (factorial (dec n_0)))))

;==(4)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (if () 1 (* n_0 (factorial (dec n_0)))))

;==(5)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (* n_0 (factorial (dec n_0))))

;==(6)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (* n_0 (factorial (dec 4))))

;==(7)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (* n_0 (factorial 3)))

;==(8)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (* n_0 ((lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))) 3)))

;==(9)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (* n_0 (if (= n_1 0) 1 (* n_1 (factorial (dec n_1))))))

;==(10)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (* n_0 (if (= 3 0) 1 (* n_1 (factorial (dec n_1))))))

;==(11)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
```

```
 (define n_0 4)
 (define n_1 3)
 (* n_0 (if () 1 (* n_1 (factorial (dec n_1))))))

;==(12)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (* n_0 (* n_1 (factorial (dec n_1)))))

;==(13)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (* n_0 (* n_1 (factorial (dec 3)))))

;==(14)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (* n_0 (* n_1 (factorial 2))))

;==(15)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (* n_0 (* n_1 ((lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))) 2))))

;==(16)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (* n_0 (* n_1 (if (= n_2 0) 1 (* n_2 (factorial (dec n_2)))))))

;==(17)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (* n_0 (* n_1 (if (= 2 0) 1 (* n_2 (factorial (dec n_2)))))))

;==(18)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (* n_0 (* n_1 (if () 1 (* n_2 (factorial (dec n_2)))))))
```

```
;==(19)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (* n_0 (* n_1 (* n_2 (factorial (dec n_2))))))

;==(20)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (* n_0 (* n_1 (* n_2 (factorial (dec 2))))))

;==(21)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (* n_0 (* n_1 (* n_2 (factorial 1)))))

;==(22)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (*
  n_0
  (* n_1 (* n_2 ((lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))) 1)))))

;==(23)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (define n_3 1)
 (* n_0 (* n_1 (* n_2 (if (= n_3 0) 1 (* n_3 (factorial (dec n_3))))))))

;==(24)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (define n_3 1)
 (* n_0 (* n_1 (* n_2 (if (= 1 0) 1 (* n_3 (factorial (dec n_3))))))))

;==(25)==>
```

```
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (define n_3 1)
 (* n_0 (* n_1 (* n_2 (if () 1 (* n_3 (factorial (dec n_3))))))))

;==(26)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (define n_3 1)
 (* n_0 (* n_1 (* n_2 (* n_3 (factorial (dec n_3)))))))

;==(27)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (define n_3 1)
 (* n_0 (* n_1 (* n_2 (* n_3 (factorial (dec 1)))))))

;==(28)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (define n_3 1)
 (* n_0 (* n_1 (* n_2 (* n_3 (factorial 0))))))

;==(29)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (define n_3 1)
 (*
  n_0
  (*
   n_1
   (* n_2 (* n_3 ((lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))) 0))))))

;==(30)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
```

```
 (define n_3 1)
 (define n_4 0)
 (*
  n_0
  (* n_1 (* n_2 (* n_3 (if (= n_4 0) 1 (* n_4 (factorial (dec n_4)))))))))

;==(31)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (define n_3 1)
 (define n_4 0)
 (* n_0 (* n_1 (* n_2 (* n_3 (if (= 0 0) 1 (* n_4 (factorial (dec n_4)))))))))

;==(32)==>
((define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
 (define n_0 4)
 (define n_1 3)
 (define n_2 2)
 (define n_3 1)
 (define n_4 0)
 (* n_0 (* n_1 (* n_2 (* n_3 (if #t 1 (* n_4 (factorial (dec n_4)))))))))

;==(33)==>
((define n_0 4) (define n_1 3) (define n_2 2)
 (define n_3 1) (* n_0 (* n_1 (* n_2 (* n_3 1)))))

;==(34)==>
((define n_0 4) (define n_1 3) (define n_2 2) (* n_0 (* n_1 (* n_2 (* 1 1)))))

;==(35)==>
((define n_0 4) (define n_1 3) (define n_2 2) (* n_0 (* n_1 (* n_2 1))))

;==(36)==>
((define n_0 4) (define n_1 3) (* n_0 (* n_1 (* 2 1))))

;==(37)==>
((define n_0 4) (define n_1 3) (* n_0 (* n_1 2)))

;==(38)==>
((define n_0 4) (* n_0 (* 3 2)))

;==(39)==>
((define n_0 4) (* n_0 6))

;==(40)==>
(* 4 6)
```

```
;==(41)==>
24
;Value 1: "Syntactic Value was returned"
```