MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1995

Supplementary Notes

A Term Rewriting Model for Scheme

Issued: Tuesday, 7 March, 1995

Substitution Models

A useful way to think about computation in Scheme is as *algebraic simplification* aimed at computing the value of an expression by successively simplifying its subexpressions. The use of names for terms is also a familiar part of arithmetic/algebraic manipulation, as in,

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

Naming rules are usually not spelled out in studying algebra, but in Scheme, as in all programming languages, such rules play a key role—arguably a more important one than the ordinary algebraic rules. 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 example, the *variable instantiation* rule is a basic naming rule:

> replace an occurrence of a variable by the expression it names.

Implicit in this rule is a simple recipe for figuring out what expression a variable names: starting at the variable to be instantiated, search back through the expression tree until you come to a definition of that variable.

In the Substitution Model sketched in 6.001 lectures and text, there is also a particular *substitution rule* for handling procedure application. In the case of a one argument procedure, the substitution rule is:

> rewrite "`((lambda (`x`) `B`) `V`)`" as "B with V substituted for x".

In these notes we describe a slightly different rule for procedure application. The Substitution Model 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[1].

We'll simply call the new rule the *lambda application rule*. In the one argument case, it 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 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 rule which allows us to remove unreferenced definitions.

Syntactic Values

We said that the point of Substitution Model rewrite rules is to arrive at the value of an expression. More precisely, the aim is to arrive at some standardized, syntactic representation called the *syntactic value* of the expression. Before examining the rules in detail, we should have a clear picture of what syntactic values look like.

The most familiar syntactic values are the *primitive values*. These are objects like numbers or truth values which have standard representations printable as output.

Scheme provides dozens of *primitive procedure variables* like `+`, `<`, `atan`, or `not` which can be applied to primitive values. These primitive procedure variables will also be syntactic values[2].

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

[1] The Substitution Model in the text 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 altogether when we introduce side-effects. Side-effects are explained in Chapter 3 of the text with a new *environment model*. You won't have to unlearn the new Substitution Model described in these Supplementary Notes, because it extends to handle side-effects quite well. In fact, the environment model can usefully be understood as a way to *implement* the new Substitution Model efficiently.

[2] We assume here that primitive procedure variables won't be redefined! In fact, according to the official definition of Revised^4 Scheme, it is possible to redefine them. In creating 6.001 Scheme from general MIT Scheme, the arithmetic operators do get redefined slightly—so that rational arithmetic does not yield unexpectedly huge quotients.

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

are called *immediate values.*

The full meaning of lambda expression like `(lambda (x) (+ x y))` isn't determined until we know what number `y` is defined to be, so we also allow syntactic values like

```
(define y 2)
(lambda (x) (+ x y))
```

which represents the "add 2" function. In general, a *syntactic value* is a sequence of zero or more *immediate definitions* followed by an immediate value, where an immediate definition is of the form `(define` ⟨variable⟩ ⟨immediate value⟩`)`.

Term Rewriting

We begin by describing rewrite rules for combinations and `if` expressions. To manage definitions, we'll also have rewrite rules for Scheme bodies, where a *body* is sequence of zero or more definitions followed by an expression. We'll use the notation

$$\langle \text{body}_1 \rangle \longrightarrow \langle \text{body}_2 \rangle$$

to indicate the rule that $\langle \text{body}_1 \rangle$ can be rewritten as, that is, replaced by, $\langle \text{body}_2 \rangle$.

When an expression not only returns a value but also causes *side-effects* like printing, then it is essential to be able to control how rules are applied, and this is a crucial aspect of Scheme programming. But as long as we focus on *values*, as we have to this point in 6.001, we needn't specify any special strategy for applying rules: a sub-body matching the lefthand side of a rule can be rewritten as the righthand side, anywhere, anytime.[3] A Substitution Model in which rules can be used anywhere is called a *term rewriting model* (TRM). That's what we will describe for Scheme.

[3] For example, Scheme unambiguously specifies that evaluation of the combination:

```
((lambda (x) ((lambda (y) 99) (display #t))) (display #f))
```

will cause the operand to be evaluated before the *body* of the operator, so `#f#t` will be printed before the value 99 is returned. On the other hand, if the inner `display` combination within the body was evaluated first, then `#t#f` would be printed before the value 99 was returned. But it's the same syntactic value, 99, returned in either case!

Primitive Application Rules

We take for granted an (infinite) set of *primitive application rules* for rewriting primitive operators on primitive values, 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

$$\texttt{(}\langle\text{primitive procedure variable}\rangle\ \langle\text{primitive value}_1\rangle\ \texttt{...}\ \texttt{)} \longrightarrow \langle\text{primitive value}\rangle.$$

By repeatedly *applying* such rules, namely replacing lefthand sides of rules with their righthand sides, we can rewrite any nested primitive combination to its primitive value. For example,

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

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

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

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

$$\begin{aligned}
\texttt{(not (< (+ 0 1 2) (/ 3 1)))} &\longrightarrow \texttt{(not (< (+ 0 1 2) 3))}\\
&\longrightarrow \texttt{(not (< 3 3))}\\
&\longrightarrow \texttt{(not \#f)}\\
&\longrightarrow \texttt{\#t}
\end{aligned}$$

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![4] For example, we might have chosen

[4] 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?)

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.

Conditional Rules

There are two rewrite rules for an `if` expression:

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

Note that, 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 immediate 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
```

Instantiation

The *variable instantiation rule* is:

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

when $\langle$variable$\rangle$ is defined to be $\langle$immediate value$\rangle$.

For example,

```
(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
```

Actually, to be used as a term rewriting rule at an arbitrary place in a body, the instantiation rule must also include a technical specification about changing defined names into "fresh" names[5]. Fortunately, this technicality can safely be ignored as long as we obey

> **The Lambda Body Rewriting Restriction**: *Don't instantiate into the body of a lambda expression.*

Real Scheme evaluation obeys the lambda body restriction.

Lambda Applications and Renaming

Scheme grammar allows definitions only in a sequence at the beginning of a body. Bodies can only appear by themselves at "top-level," or as the bodies of lambda expressions[6].

When the lambda application rule creates new immediate definitions[7], we just stick them anyplace that is grammatical. Definitions at the beginning of a lambda body may also have to be moved after the lambda disappears. So the form of the lambda application rule is:

`((lambda (`$\langle$variable$_1\rangle$ `...` `)` $\langle$defines$\rangle$ $\langle$expression$\rangle$`))` $\langle$immediate value$_1\rangle$ `...` `)` $\longrightarrow$ $\langle$expression$\rangle$

> with the sequence
>
> `(define` $\langle$variable$_1\rangle$ $\langle$immediate value$_1\rangle$`)`$\ldots\langle$defines$\rangle$
>
> placed in the body above $\langle$expression$\rangle$ at any position at which a sub-body may begin.

[5]The problem is that after instantiation, there are two copies of $\langle$immediate value$\rangle$—the original one in the definition of $\langle$variable$\rangle$, and another at the place where $\langle$variable$\rangle$ was instantiated. It may happen that some variable *within* $\langle$immediate value$\rangle$ winds up underneath a different definition in the second copy than it did in the original. This has to be prevented in order to mantain the requirement that Scheme be statically scoped.

[6]A formal grammar for TRM Scheme is attached as an Appendix for your reference.

[7]To obtain "call-by-name," or "normal order," behavior, we must relax the condition that the operands of the combination be $\langle$immediate value$\rangle$'s, allowing them instead to be arbitrary expressions.

But watch out! This time there is a naming problem we cannot avoid.

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 static scoping is to nsure that names of formals or names defined within an expression can be "kept private" so they behave independently 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.

For example, 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)
```

As the names suggest, `add2` has the behavior of adding two to its operand; similarly, `add3` adds three. 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 `make-incrementer`:

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

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

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

Lambda application comes next, yielding:

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

Now there is an obvious problem, because we have two definitions of `n` at the beginning of the block, which is ambiguous and indeed violates Scheme grammar. We handle this by renaming the newly defined `n` and the occurrences it defines, 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 bound to which value becomes unresolvable[8].

So a more accurate statement of the *lambda application rule* is:

`((lambda (`$\langle$variable$_1\rangle$ `... )` $\langle$defines$\rangle\langle$expression$\rangle$`)` $\langle$immediate value$_1\rangle$ `... )` $\longrightarrow$ $\langle$expression$\rangle$

> with the sequence
>
> `(define` $\langle$variable$_1\rangle$ $\langle$immediate value$_1\rangle$`)` $\ldots\langle$defines$\rangle$
>
> placed above $\langle$expression$\rangle$ in any position in the body at which a sub-body may begin, **with defined variables freshly renamed as necessary.**

[8] 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."

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 TRM is dropping unreferenced definitions, so we call this the *garbage collection rule*:

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

A subset G of immediate definitions among the 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.

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))
```

Uniqueness

The full set of term rewriting rules has same the unique value property as the simple algebraic rules:

> **The Unique Value Theorem**. *There is at most one primitive value that can be reached by successively rewriting a body using the Scheme Term Rewriting rules above.*

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 such as the alternative branch of an `if` whose test evaluates to `#t`.

We also observed that real Scheme evaluation obeys the lambda body restriction. 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 Term Rewriting 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 while obeying 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.) There is a whole subdiscipline of Theoretical Computer Science called Term Rewriting Theory which studies such properties.

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`.

Instead of extending the 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

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

So we can specify the computations of derived expressions by translating, or "desugaring," them into kernel ⟨expression⟩'s[9]. The Revised[4] Scheme Report describes the translations above and others for ⟨and⟩'s, ⟨or⟩'s, and several further forms.

Lists and Symbols

Scheme output notation for lists and symbols makes them look just like expressions. Using this notation in the TRM presents a problem: how do we tell the difference between lists and combinations[10]? 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[11]. Namely, we add the appropriate procedure variables such as `cons`, `car` and `cdr`, and regard any expression of the form

$$\texttt{(cons}\ \langle\text{immediate value}\rangle\ \langle\text{immediate value}\rangle\texttt{)}$$

as an 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`:

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

---

[9] 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.

[10] 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.

[11] This way of handling list works usefully until we begin "mutating" lists, at which point some more refined description of 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 $\langle\text{immediate value}_3\rangle$ is not of the form $\texttt{(cons } \langle\text{immediate value}_1\rangle\ \langle\text{immediate value}_2\rangle\texttt{)}$

$$\texttt{(null? ())} \longrightarrow \texttt{\#t}$$

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

if ⟨immediate value⟩ is not `()`.

$$\texttt{(list)} \longrightarrow \texttt{()}$$

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

Running the Substitution Model

There is an implementation of the Substitution Model which you are welcome to use. 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.

In the 6.001 Edwin editor, use `M-x load-problem-set: 4` to load the code for the Substitution Model. You can generate a list of the steps in the rewriting of an expression (in reverse order) by applying the procedure `smstep-list` to the quoted expression. The order in which rewrite rules are applied reflects Scheme evaluation order fairly accurately. To generate the list and print it nicely, apply the procedure `smeval` to the quoted expression.

For example, evaluate

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

to see expressions at selected steps as this lambda application successively rewrote 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 (rec-factorial n)
```

```
      (if (<= n 0)
          1
          (* n (rec-factorial (dec n)))))
    (rec-factorial 6)))
```

will allow you to watch how this body rewrote to `720`.

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[12].

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. To be able to see the final value in more familiar format, you should evaluate

```
(define reversed-steps (smstep-list body))

(define final-body (body-or-info-of-step (car reversed-steps)))

(pp (printable-version final-body))
```

The procedure `save-this-step?` determines which steps get saved on the list generated by `smstep-list` for printing by `smeval`. The default is to print more sparsely as the step-number grows. To save every step, evaluate

```
(define (save-this-step? step-number body) #t)
```

The procedure `garbage-collect-this-step?` can be used to force more frequent garbage collections. Its default arbitrarily imposes a garbage collection every fortieth step. To avoid such extra garbage collection (which can slow things down a bit), evaluate

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

[12] 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

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

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 `smstep-list` generates and view it selectively.

Bugs

We don't know of any yet, but there surely are some. Gratitude and lots of 6.001 brownie points for first bug reports—email them to `6001-lecturers@ai.mit.edu`.

Appendix: Grammar for Functional Scheme

Kernel Syntax

⟨define⟩ := `(define` ⟨variable⟩ ⟨expression⟩ `)`
⟨body⟩ ::= ⟨define$_1$⟩ . . . ⟨expression⟩
(Note: all defined variables must be distinct)

⟨expression⟩ ::= ⟨variable⟩ | ⟨self-evaluating⟩ | ⟨symbol⟩ | `()`
| ⟨combination⟩ | ⟨lambda expression⟩ | ⟨if⟩

⟨self-evaluating⟩ ::= ⟨numeral⟩ | ⟨boolean⟩
⟨numeral⟩ ::= `0` | `-1` | `3.14159` | . . .
⟨boolean⟩ ::= `#t` | `#f`

⟨symbol⟩ ::= `(quote` ⟨variable⟩ `)` | `(quote` ⟨keyword⟩`)`

⟨combination⟩ ::= `(` ⟨operator⟩ ⟨operand$_1$⟩ . . . `)`
⟨operator⟩ ::= ⟨expression⟩
⟨operand⟩ ::= ⟨expression⟩

⟨lambda expression⟩ ::= `(lambda (`⟨formals⟩`)` ⟨body⟩ `)`
⟨formals⟩ ::= ⟨variable$_1$⟩ . . .
(Note: all ⟨variable⟩'s must be distinct)

⟨if⟩ := `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternative⟩ `)`
⟨test⟩ ::= ⟨expression⟩
⟨consequent⟩ ::= ⟨expression⟩
⟨alternative⟩ ::= ⟨expression⟩

⟨primitive procedure variable⟩ ::= `+` | `-` | `*` | `/` | `=` | `<` | `atan` | . . .
⟨rule-specified procedure variable⟩ ::= `null?` | `pair?` | `car` | `cdr` | `cons` | `list` | `equal?` | `apply`
⟨keyword⟩ ::= `define` | `quote` | `lambda` | `if`
⟨variable⟩ ::= identifiers which are not ⟨self-evaluating⟩ or ⟨keyword⟩'s

Derived Syntax

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

Values

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