Massachusetts Institute of Technology
6.844, Spring '05: Computability Theory of and with Scheme
Prof. Albert R. Meyer

Course Notes 6
March 14
revised March 14, 2005, 1362 minutes

A Substitution Model for Scheme, I: Rewrite Rules

1 Introduction

These notes describe a Scheme Substitution Model: an accurate, simple mathematical model of Scheme evaluation based on rules for rewriting one Scheme expression into another. The model captures a significant portion of Scheme, including asssignment (`set!`) and control abstraction (`call/cc`).

We assume the reader already has an understanding of Scheme at the level taught in an introductory Scheme programming course. In particular, we assume the concepts of free and bound variables, and the scoping rules for `lambda` and `letrec`, are understood.

The rules of the game in a Substitution Model are that the only objects manipulated in the Model are Scheme expressions: no separate data structures for environments, `cons`-cells, or continuations. Evaluation of an expression, M, is modelled by successive application of rewrite rules starting with M. Each rule transforms an expression into a new Scheme expression. Rewriting continues until an expression is reached for which no rule is applicable. This final expression, if any, gives a direct representation either of the value returned by the expression, or of the kind of dynamic error that first occurs in the evaluation.

At most one rule is applicable to each expression, reflecting the deterministic character of Scheme evaluation. The way an expression rewrites is determined solely by the expression, not the sequence of prior rewrites that may have led to it. If M' is an expression reached at any point by rewriting starting at M, then evaluating M' in Scheme's initial environment will result in the same final value, the same kind of error, or the same "runaway" behavior (divergence) as evaluation of M.

The environment in which an expression is to be evaluated will be represented by surrounding an expression with an outermost `letrec` that binds variables in the environment to the expressions representing their values. Immutable lists and pairs are represented as combinations with operators `list` or `cons`.

Mutable lists do not fit well into a Substitution Model. They could be shoe-horned in, but we haven't found a tasteful way to do it. The problem is that we haven't found a reasonable class of Scheme expressions that evaluate directly to circular list structures and that could serve as canonical forms for these structures. So we have omitted mutable lists from this Substitution Model; vectors are omitted for similar reasons. We have omitted characters altogether, ensuring that strings are immutable.

Side-effects involving input/output also don't fit and have been omitted. So procedures such as `set-car!`, `string-set!`, `display`, or `read` are not included in the Model.

Copyright © 2005, Prof. Albert R. Meyer. All rights reserved.

2 Control Syntax for the Substitution Model

A simplified Backus-Naur Form (BNF) grammar for Scheme is given in an Appendix. According to the official Scheme specification, the standard builtin operators such as `+`, `symbol?`, `cons`, `apply` are variables that can be reassigned. This is a regrettable design decision, because with few exceptions, it is a really bad idea for a programmer to redefine the builtins. In this Substitution Model, these identifiers are treated as *constants* rather than variables.

The Substitution Model requires some additional syntactic concepts, namely, *syntactic values* and *control contexts*.

2.1 Syntactic Values

Scheme's values are numbers, Booleans, symbols, and similar atomic types; procedures; and lists and pairs of values. Each value will be represented by a canonical expression called a *syntactic value*. In particular, compound procedures are represented as `lambda`-expressions. Here is the grammar[1]:

$$\begin{array}{rcl}
\langle\text{syntactic-value}\rangle & ::= & \langle\text{immediate-value}\rangle \mid \langle\text{list-value}\rangle \mid \langle\text{nonlist-pair-value}\rangle \\
\langle\text{immediate-value}\rangle & ::= & \langle\text{self-evaluating}\rangle \mid \langle\text{symbol}\rangle \mid \langle\text{procedure}\rangle \\
\langle\text{list-value}\rangle & ::= & \texttt{(list } \langle\text{syntactic-value}\rangle^{*}\texttt{)} \\
\langle\text{nonlist-pair-value}\rangle & ::= & \texttt{(cons } \langle\text{syntactic-value}\rangle\ \langle\text{immediate-value}\rangle\texttt{)} \\
 & & \mid \texttt{(cons } \langle\text{syntactic-value}\rangle\ \langle\text{nonlist-pair-value}\rangle\texttt{)}
\end{array}$$

Note that syntactic values may contain `letrec`'s only within procedure bodies.

Problem 1. **(a)** Verify that an expression, M, is a ⟨nonlist-pair-value⟩ iff M is a ⟨syntactic-value⟩ not of the form `(list` ⟨syntactic-value⟩*`)`.

(b) Verify that an expression, M, is a ⟨syntactic-value⟩ that is not of the form `(list` ⟨syntactic-value⟩*`)` or `(cons ...)` iff M is an ⟨immediate-value⟩.

2.2 Control Contexts for Kernel Scheme

An important technical property of the Substitution Model is that the rewrite rule to apply at any evaluation step wiil be uniquely determined. The order in which subexpressions are evaluated is formalized in terms of *control contexts*. A control context is an expression with a "hole," [ ], indicating the subexpression that an evaluator would begin working on. We'll illustrate with an example before giving the formal definitions.

[1] In these grammars, superscript "*" indicates zero or more occurrences of a grammatical phrase, and superscript "$^+$" indicates one or more occurrences.

Definition 2.1. If R is an expression with a hole, and M is an expression, we write $R[M]$ to denote the result of replacing the hole in R by M *without any renaming* of bound variables.

Example 2.2. Let

$M =$ `(+ 1 (if (pair? (list (list) 'a)) 2 3) (* 4 5))`.

M is a combination, and not all the operands are values, so Scheme would start to evaluate one of the operands. Using left-to-right evaluation, the operand

`(if (pair? (list (list) 'a)) 2 3)`

would be the one to start evaluating, since `+` and `1` represent final values. This corresponds to parsing M as

R_1[`(if (pair? (list (list) 'a)) 2 3)`]

where R_1 is the control context

$R_1 ::=$ `(+ 1 [ ] (* 4 5))`.

The test of this `if` expression is a combination

$P ::=$ `(pair? (list (list) 'a))`

whose operator and operand are values, so next, Scheme would actually apply the operator `pair?` to the operand `(list (list) 'a)`. This corresponds parsing M as $R[P]$ where

$R ::=$ `(+ 1 (if [ ] 2 3) (* 4 5))`.

The fact that Scheme will apply the operator `pair?` is captured by the fact that P is an *immediate redex*.

The fact that the rewrite rule to apply at any evaluation step is uniquely determined follows from the fact that every nonvalue Scheme expression parses in a unique way as a control context with an immediate redex in its hole.

Formally, we specify control contexts and immediate redexes by the following grammars:

```
⟨control-context⟩  ::=  ⟨hole⟩
                         | (if ⟨control-context⟩ ⟨expression⟩ ⟨expression⟩)
                         | (begin ⟨control-context⟩ ⟨expression⟩⁺)
                         | (set! ⟨variable⟩ ⟨control-context⟩)
                         | (⟨let-keyword⟩
                              (⟨value-binding⟩* (⟨variable⟩ ⟨control-context⟩) ⟨binding⟩*)
                                ⟨expression⟩)
                         | (⟨syntactic-value⟩* ⟨control-context⟩ ⟨expression⟩*)
            ⟨hole⟩  ::=  [ ]
   ⟨value-binding⟩  ::=  (⟨variable⟩ ⟨syntactic-value⟩)
```

Note that `letrec`'s all of whose ⟨init⟩'s are syntactic values may only appear in control contexts when they are within procedure bodies.

Problem 2. Verify that if R_1 and R_2 are control contexts, then so is $R_1[R_2]$.

$$
\begin{array}{lcl}
\langle\text{immediate-redex}\rangle & ::= & \langle\text{variable}\rangle \\
 & & \mid \texttt{(if}\ \langle\text{syntactic-value}\rangle\ \langle\text{expression}\rangle\ \langle\text{expression}\rangle\texttt{)} \\
 & & \mid \texttt{(}\langle\text{let-keyword}\rangle\ \texttt{(}\langle\text{value-binding}\rangle^*\texttt{)}\ \langle\text{expression}\rangle\texttt{)} \\
 & & \mid \texttt{(}\langle\text{nonpairing-procedure}\rangle\ \langle\text{syntactic-value}\rangle^*\texttt{)} \\
 & & \mid \texttt{(begin}\ \langle\text{expression}\rangle\texttt{)} \\
 & & \mid \texttt{(begin}\ \langle\text{syntactic-value}\rangle\ \langle\text{expression}\rangle^*\texttt{)} \\
 & & \mid \texttt{(set!}\ \langle\text{variable}\rangle\ \langle\text{syntactic-value}\rangle\texttt{)}
\end{array}
$$

Definition 2.3. An outermost `letrec` binding variables to values—used to model a Scheme environment—is called the *environment* `letrec`. An expression is said to be in *environment form* when it has an environment `letrec`, namely, it is of the form

$$\texttt{(letrec (}\langle\text{value-binding}\rangle^*\texttt{)}\ N\texttt{)}$$

for some ⟨expression⟩, N. We use $\text{Env}(N)$ as an abbreviation for this form.

Definition 2.4. Let M be a Scheme expression, R be a control context, and P be an immediate redex. Then M is said to *control-parse into R and P* iff either

1. M is of the form $\text{Env}(R[P])$, or
2. $M = R[P]$ for $R \neq \langle\text{hole}\rangle$, or
3. $M = P$, $R = \langle\text{hole}\rangle$, and P is not in environment form .

Definition 2.4.3 reflects that fact that a non-outermost `letrec` binding of variables to values will be the redex of a rule to incorporate the bindings into the outermost environment `letrec`. On the other hand, we do not want to parse the environment `letrec` in this way. For example, consider the expression

$$\texttt{(letrec ((x 1)) (letrec ((y 2)) (+ x y))).} \tag{1}$$

Expression (1) control parses into

$$
\begin{aligned}
R &= \texttt{(letrec ((x 1))}\ [\ ]\texttt{)}, \\
P &= \texttt{(letrec ((y 2)) (+ x y))}.
\end{aligned}
$$

On the other hand, (1) is itself is an ⟨immediate-redex⟩ according to the grammatical rules, so it could also be parsed as $R'[P']$, where $R' = \langle\text{hole}\rangle$, and P' is (1) itself. However, Definition 2.4.3 disallows this second parse as a control parse because P' is in environment form.

Lemma 2.5. *(**Unique Control Parsing**) If a Scheme expression, M, is not a syntactic value, then there is a unique control context, R, and immediate redex, P, such that $M = R[P]$. If M is a syntactic value, then it is not control-parsable.*

Proof. By structural induction on M. If M is:

- [⟨self-evaluating⟩, ⟨symbol⟩, or ⟨procedure⟩] In this case M is a ⟨syntactic-value⟩, and we must show that it cannot be parsed as $R[P]$. But it follows immediately from the definitions of ⟨syntactic-value⟩ and ⟨control-context⟩, that the only control context R such that $M = R[N]$ for some N, is $R = [\,]$, in which case $N = M$. But since $N = M$ is an ⟨syntactic-value⟩, it is not an ⟨immediate-redex⟩, proving that M is not control-parsable.
- [a variable] In this case $R = \langle\text{hole}\rangle$ and $P = M$.
- [a combination] Then if:
  - [the operator or some operand is not a ⟨syntactic-value⟩] Then M is of the form

    $$(V_0 \ldots\ N\ M_0 \ldots)$$

    where $V_0 \ldots$ is a (possibly empty) sequence of syntactic values, N is not a syntactic value, and $M_0 \ldots$ is a (possibly empty) sequence of expressions. In this case, M is neither a syntactic value nor an immediate redex. Now it follows immediately from the definitions of ⟨syntactic-value⟩ and ⟨control-context⟩, that any control context R such that $M = R[M']$ for some M', must be of the form $R = (V_0 \ldots\ R'\ M_0 \ldots)$ for some ⟨control-context⟩, R' such that $R'[M'] = N$. But by induction, $N = R'[P]$ for a unique ⟨control-context⟩, R', and ⟨immediate-redex⟩, P. Hence, $M = R[P]$ for these uniquely determined R and P.
  - [the operator and all operands are ⟨syntactic-value⟩'s] Then M is of the form $(op\ V_0 \ldots)$ where $V_0 \ldots$ is a (possibly empty) sequence of syntactic values and *op* is a syntactic value. By induction, there is no ⟨control-context⟩, R' such that $R'[P] = op$ or $R'[P] = V_i$ for some immediate redex P and operand V_i. Now it follows immediately from the definitions of ⟨syntactic-value⟩ and ⟨control-context⟩, that the only control context, R, such that $M = R[P]$ for some immediate redex, P, must be with $R = \langle\text{hole}\rangle$ and $P = M$. If *op* is not a ⟨pairing-operator⟩, then $M = P$ is an immediate redex, and R and P are uniquely determined as required. On the other hand, if *op* is a ⟨pairing-operator⟩, then M is a value, not an immediate redex, so M is not control-parsable, as required in this case.
- [etc.] The remaining cases are similar.

□

3 Scheme Rewrite Rules

This section contains all the rewrite rules necessary to specify the evaluation of kernel Scheme expressions.

We will not consider rewrite rules for the derived expressions. The Revised^5 Scheme Manual describes how to translate ("desugar") expressions using derived syntax into Kernel Scheme. These translations can easily be described with rewrite rules, and these desugaring rules could be incorporated directly into a Substitution Model. The desugaring rules raise no new issues beyond those we consider for the kernel rules, so we have omitted them.

In the following sections, R denotes a control context, B denotes a sequence of zero or more ⟨value-binding⟩'s of distinct variables, V denotes a syntactic value, $V_1 \ldots$ a sequence of one or more syntactic values, and x denotes a variable.

3.1 Simple Control Rules

Definition 3.1. A *simple control rule* is a rewrite rule of the form

$$R[P] \to R[T]$$

or

$$\texttt{(letrec (}B\texttt{) } R[P]\texttt{)} \to \texttt{(letrec (}B\texttt{) } R[T]\texttt{)}.$$

Such a pair of simple control rules may be abbreviated as $P \to T$, showing only the subexpressions that are changed by the rule. In this case, P will be an immediate redex and is called the *immediate redex of the rule*, and T is called the *immediate contractum*.

3.1.1 Rules for Kernel Scheme

The redexes and contracta for the kernel Scheme simple control rules are:

- `if`:

$$\begin{aligned} \texttt{(if \#f } M\ N\texttt{)} &\to N \\ \texttt{(if } V\ M\ N\texttt{)} &\to M, \qquad \text{for } V \neq \texttt{\#f} \end{aligned}$$

- `lambda` *no args:*

$$\texttt{((lambda () } M\texttt{))} \to M$$

- `begin`:

$$\begin{aligned} \texttt{(begin } M\texttt{)} &\to M \\ \texttt{(begin } V\ \langle\text{expression}\rangle^{+}\texttt{)} &\to \texttt{(begin } \langle\text{expression}\rangle^{+}\texttt{)} \end{aligned}$$

- `procedure?`:

$$\begin{aligned} \texttt{(procedure? } V\texttt{)} &\to \texttt{\#t}, \qquad \text{for } V \text{ a } \langle\text{procedure}\rangle, \\ \texttt{(procedure? } V\texttt{)} &\to \texttt{\#f}, \qquad \text{for other } V. \end{aligned}$$

- *builtin operations:*

$$\begin{aligned} \texttt{(+ 2 3)} &\to \texttt{5} \\ \texttt{(string-append "ab" "cde")} &\to \texttt{"abcde"} \\ \texttt{(boolean? "ab")} &\to \texttt{\#f} \\ &\ \ \vdots \end{aligned}$$

- *symbols:*

$$\begin{aligned} \texttt{(symbol? (quote } S\texttt{))} &\to \texttt{\#t} \\ \texttt{(symbol? } V\texttt{)} &\to \texttt{\#f}, \quad \text{if } V \text{ is not } \texttt{(quote } S\texttt{)} \\ \texttt{(eq? (quote } S\texttt{) (quote } S\texttt{))} &\to \texttt{\#t} \\ \texttt{(eq? (quote } S_1\texttt{) (quote } S_2\texttt{))} &\to \texttt{\#f}, \quad \text{if } S_1 \neq S_2, \end{aligned}$$

  where S is an $\langle$identifier$\rangle$.

- *lists:*

$$\begin{array}{rcl}
\texttt{(cons } V\ \langle\text{nil}\rangle\texttt{)} & \rightarrow & \texttt{(list } V\texttt{)} \\
\texttt{(cons } V\ \texttt{(list } V_1 \ldots\texttt{))} & \rightarrow & \texttt{(list } V\ V_1 \ldots\texttt{)} \\
\texttt{(car (list } V_1 \ldots\texttt{))} & \rightarrow & V_1 \\
\texttt{(cdr (list } V_1\ V_2\ \ldots\texttt{))} & \rightarrow & \texttt{(list } V_2 \ldots\texttt{)} \\
\texttt{(null? } \langle\text{nil}\rangle\texttt{)} & \rightarrow & \texttt{\#t} \\
\texttt{(null? } V\texttt{)} & \rightarrow & \texttt{\#f}, \quad \text{if } V \neq \langle\text{nil}\rangle \\
\texttt{(pair? (list } \langle\text{syntactic-value}\rangle^{+}\texttt{))} & \rightarrow & \texttt{\#t} \\
\texttt{(pair? (list))} & \rightarrow & \texttt{\#f} \\
\texttt{(pair? } \langle\text{immediate-value}\rangle\texttt{)} & \rightarrow & \texttt{\#f} \\
\texttt{(apply } V\ \langle\text{nil}\rangle\texttt{)} & \rightarrow & \texttt{(}V\texttt{)} \\
\texttt{(apply } V\ \texttt{(list } V_1 \ldots\texttt{))} & \rightarrow & \texttt{(}V\ V_1 \ldots\texttt{)}
\end{array}$$

- *pairs:*

$$\begin{array}{rl}
\texttt{(pair? (cons } V_1\ K\texttt{))} & \rightarrow \texttt{\#t} \\
\texttt{(car (cons } V_1\ K\texttt{))} & \rightarrow V_1 \\
\texttt{(cdr (cons } V_1\ K\texttt{))} & \rightarrow K
\end{array}$$

where K is a ⟨nonlist-pair-value⟩.

This condition that K be a ⟨nonlist-pair-value⟩ instead of simply a ⟨syntactic-value⟩ is included for technical reasons, namely, to ensure that at most one rewrite rule applies to any expression[2] (*cf.*, Corollary 3.3 below).

3.2 Environment Rules

The following rules for Kernel Scheme update the environment `letrec`.

[2]For example, by the given rules, the expression `(pair? (cons` V_1 `(list)))` rewrites to `(pair? (list` V_1`))` which then rewrites to `#t`. Relaxing the condition that K be a ⟨nonlist-pair-value⟩ to be only that K be a ⟨syntactic-value⟩ would allow `(pair? (cons` V_1 `(list)))` also to rewrite *directly* `#t`. While this would still correctly model Scheme evaluation, it would violate the technically convenient property that at most one rewrite rule applies to any expression.

- `lambda` *bind an arg:*

$$\begin{aligned}&\texttt{(letrec (}B\texttt{)}\ R[\texttt{((lambda (}x_1\ \texttt{...}\ \texttt{)}\ M\texttt{)}\ V_1\cdots\texttt{)}]\texttt{)}\\&\quad\to\ \texttt{(letrec (}B\ \texttt{(}x_1\,V_1\texttt{))}\ R[\texttt{((lambda (}\ldots\texttt{)}\ M\texttt{)}\ \cdots\texttt{)}]\texttt{)},\end{aligned}$$

$$\begin{aligned}&R[\texttt{((lambda (}x_1\ \texttt{...}\ \texttt{)}\ M\texttt{)}\ V_1\cdots\texttt{)}]\\&\quad\to\ \texttt{(letrec ((}x_1\,V_1\texttt{))}\ R[\texttt{((lambda (}\ldots\texttt{)}\ M\texttt{)}\ \cdots\texttt{)}]\texttt{)}.\end{aligned}$$

$$\begin{aligned}&\texttt{(letrec (}B\texttt{)}\ R[\texttt{((lambda}\ x\ M\texttt{)}\ V_1\ldots\texttt{)}]\texttt{)}\\&\quad\to\ \texttt{(letrec (}B\ \texttt{(}x\ \texttt{(list}\ V_1\ldots\texttt{)))}\ R[M]\texttt{)},\end{aligned}$$

$$\begin{aligned}&R[\texttt{((lambda}\ x\ M\texttt{)}\ V_1\ldots\texttt{)}]\\&\quad\to\ \texttt{(letrec ((}x\ \texttt{(list}\ V_1\ldots\texttt{)))}\ R[M]\texttt{)}.\end{aligned}$$

- *nested* `letrec`*:*

$$\begin{aligned}R[\texttt{(letrec (}B_2\texttt{)}\ M\texttt{)}]\ &\to\ \texttt{(letrec (}B_2\texttt{)}\ R[M]\texttt{)}\qquad\text{for } R\neq\langle\text{hole}\rangle,\\\texttt{(letrec (}B_1\texttt{)}\ R[\texttt{(letrec (}B_2\texttt{)}\ M\texttt{)}]\texttt{)}\ &\to\ \texttt{(letrec (}B_1\,B_2\texttt{)}\ R[M]\texttt{)}.\end{aligned}$$

- *instantiation:*

$$\texttt{(letrec (}B_1\ \texttt{(}x\,V\texttt{)}\ B_2\texttt{)}\ R[x]\texttt{)}\ \to\ \texttt{(letrec (}B_1\ \texttt{(}x\,V\texttt{)}\ B_2\texttt{)}\ R[V]\texttt{)}$$

- *assignment:*

$$\begin{aligned}&\texttt{(letrec (}B_1\,\texttt{(}x\,V_1\texttt{)}\,B_2\texttt{)}\,R[\texttt{(set!}\,x\,V_2\texttt{)}]\texttt{)}\\&\quad\to\ \texttt{(letrec (}B_1\,\texttt{(}x\,V_2\texttt{)}\,B_2\texttt{)}\,R[\texttt{(quote set!-done)}]\texttt{)}\end{aligned}$$

3.3 Unique Rewriting

Definition 3.2. For Scheme expressions M, N, we write $M \to N$ to indicate that M rewrites to N by *one application* of a Scheme Substitution Model rewrite rule. $M\not\to$ means that no rewrite rule applies to M.

From the form of the Substitution Model rewrite rules and the Unique Control Parsing Lemma 2.5, we can straightforwardly conclude:

Corollary 3.3. *(**Unique Rewriting**) There is at most one Scheme Substitution Model rewrite rule whose pattern matches an expression M, and if there is such a rewrite rule, its match is unique. Hence, for every Scheme expression, M, there is at most one N such that $M \to N$.*

Scheme's evaluation behavior is "sequential." Namely, if in the process of evaluation, a subexpression starts to be evaluated, then evaluation continues "at that subexpression" until a value for the subexpression is returned. This happens regardless of how evaluation would proceed once a value is returned. In particular, no Scheme evaluation would switch back and forth between disjoint subexpressions to evaluate them in parallel. The Control-context Independence Corollary makes this precise.

Corollary 3.4. *(Control-context Independence)*

1. *If M_2 is* not *in environment form then*

$$M_1 \rightarrow M_2 \quad \textit{implies} \quad R[M_1] \rightarrow R[M_2].$$

2. *If*

$$\texttt{(letrec (}B_1\texttt{) }M_1\texttt{)} \rightarrow \texttt{(letrec (}B_2\texttt{) }M_2\texttt{)}$$

then

$$\texttt{(letrec (}B_1\texttt{) }R[M_1]\texttt{)} \rightarrow \texttt{(letrec (}B_2\texttt{) }R[M_2]\texttt{)}.$$

3. *If M_1 is not in environment form and*

$$M_1 \rightarrow \texttt{(letrec (}B\texttt{) }M_2\texttt{)},$$

then

$$R[M_1] \xrightarrow{\leq 2} \texttt{(letrec (}B\texttt{) }R[M_2]\texttt{)}$$

where $\xrightarrow{\leq 2}$ *indicates successive application of at most two rewriting rules.*

An example of Corollary 3.4.3 where two rule applications are needed is when

$$\begin{aligned} M_1 &= \texttt{(letrec ((x 1)) (+ x x))}, \\ R &= \texttt{(- [ ])}. \end{aligned}$$

In this case, for

$$\begin{aligned} B &= \texttt{(x 1)}, \\ M_2 &= \texttt{(+ 1 x)}, \end{aligned}$$

we have $M_1 \rightarrow \texttt{(letrec (}B\texttt{) }M_2\texttt{)}$, but it takes 2 steps to rewrite $R[M_1]$ to the desired form:

$$\begin{aligned} R[M_1] &= \texttt{(- (letrec ((x 1)) (+ x x)))} \\ &\rightarrow \texttt{(letrec ((x 1)) (- (+ x x)))} \qquad \text{(by the nested \texttt{letrec} rule)} \\ &\rightarrow \texttt{(letrec ((x 1)) (- (+ 1 x)))} \\ &= \texttt{(letrec (}B\texttt{) }R[M_2]\texttt{)}. \end{aligned}$$

Problem 3. Prove Corollary 3.4. *Hint:* Use Problem 2 and Unique Control Parsing.

A Scheme Syntax in BNF

The following Backus-Naur Form (BNF) grammars describe the main constructs of Scheme[3]. In these grammars, superscript "*" indicates zero or more occurrences of a grammatical phrase, and superscript "$^+$" indicates one or more occurrences.

[3]For the official, full Scheme grammar, see the Revised^5 Scheme Manual available on the web at:

`http://www.schemers.org/Documents/Standards/R5RS`

A.1 The Functional Kernel

$$
\begin{array}{rcl}
\langle\text{expression}\rangle & ::= & \langle\text{self-evaluating}\rangle \mid \langle\text{symbol}\rangle \mid \langle\text{variable}\rangle \\
 & & \mid \langle\text{procedure}\rangle \mid \langle\text{let-form}\rangle \mid \langle\text{if}\rangle \mid \langle\text{combination}\rangle \\
 & & \\
\langle\text{self-evaluating}\rangle & ::= & \langle\text{numeral}\rangle \mid \langle\text{boolean}\rangle \mid \langle\text{string}\rangle \mid \ldots \\
\langle\text{numeral}\rangle & ::= & \texttt{0} \mid \texttt{-1} \mid \texttt{314159} \mid \ldots \\
\langle\text{boolean}\rangle & ::= & \texttt{\#t} \mid \texttt{\#f} \\
\langle\text{string}\rangle & ::= & \texttt{"hello there"} \mid \ldots \\
 & & \\
\langle\text{symbol}\rangle & ::= & \texttt{(quote}\ \langle\text{identifier}\rangle\texttt{)} \\
\langle\text{identifier}\rangle & ::= & \text{identifiers that are not } \langle\text{self-evaluating}\rangle \\
\langle\text{variable}\rangle & ::= & \text{identifiers that are neither } \langle\text{self-evaluating}\rangle\text{, } \langle\text{keyword}\rangle \\
 & & \langle\text{procedure-constant}\rangle\text{, nor } \langle\text{pairing-operator}\rangle \\
 & & \\
\langle\text{keyword}\rangle & ::= & \texttt{quote} \mid \texttt{lambda} \mid \langle\text{let-keyword}\rangle \mid \texttt{if} \\
 & & \\
\langle\text{procedure}\rangle & ::= & \langle\text{nonpairing-procedure}\rangle \\
\langle\text{nonpairing-procedure}\rangle & ::= & \langle\text{lambda-expression}\rangle \mid \langle\text{procedure-constant}\rangle \\
\langle\text{lambda-expression}\rangle & ::= & \texttt{(lambda (}\langle\text{formals}\rangle\texttt{)}\ \langle\text{expression}\rangle\texttt{)} \\
\langle\text{formals}\rangle & ::= & \langle\text{variable}\rangle^{*} \qquad \text{(Note: all } \langle\text{variable}\rangle\text{'s must be distinct.)} \\
\langle\text{procedure-constant}\rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{=} \mid \texttt{<} \mid \texttt{atan} \mid \texttt{string=?} \mid \ldots \\
 & & \mid \texttt{number?} \mid \texttt{symbol?} \mid \texttt{procedure?} \mid \texttt{string?} \mid \texttt{boolean?} \mid \texttt{eq?} \mid \ldots
\end{array}
$$

Note that no "side-effect" procedures such as `display`, `set-car!`, `string-set!` nor "pairing" operators `list`, `cons` are included among the procedure constants. Also, as a further reflection of our explanation why mutable lists have been omitted from the Substitution Model, we restrict application of `eq?` to values that are ⟨symbol⟩'s.

⟨let-form⟩	::=	`(`⟨let-keyword⟩ `(`⟨binding⟩* `)` ⟨expression⟩`)`
		(Note: all variables bound by the ⟨let-form⟩ must be distinct.)
⟨let-keyword⟩	::=	`letrec`
⟨binding⟩	::=	`(`⟨variable⟩ ⟨init⟩`)`
⟨init⟩	::=	⟨expression⟩
⟨if⟩	::=	`(if` ⟨test⟩ ⟨consequent⟩ ⟨alternative⟩`)`
⟨test⟩	::=	⟨expression⟩
⟨consequent⟩	::=	⟨expression⟩
⟨alternative⟩	::=	⟨expression⟩
⟨combination⟩	::=	`(`⟨operator⟩ ⟨operand⟩* `)`
⟨operator⟩	::=	⟨expression⟩
⟨operand⟩	::=	⟨expression⟩

A.2 Functional (Immutable) Lists

⟨expression⟩	::=	... | ⟨nil⟩
⟨nil⟩	::=	`(list)`
⟨procedure⟩	::=	... | ⟨pairing-operator⟩
⟨pairing-operator⟩	::=	`cons` | `list`
⟨procedure-constant⟩	::=	... | `car` | `cdr` | `map` | `apply` | `null?` | `pair?`
		(Note: `cons` and `list` are *not* considered to be procedure constants.)
⟨lambda-expression⟩	::=	... | `(lambda` ⟨variable⟩ ⟨expression⟩`)`

A.3 The Full Kernel

⟨expression⟩	::=	... | ⟨begin⟩ | ⟨assignment⟩
⟨keyword⟩	::=	... | `begin` | `set!`
⟨begin⟩	::=	`(begin` ⟨expression⟩+ `)`
⟨assignment⟩	::=	`(set!` ⟨variable⟩ ⟨expression⟩`)`

A.4 Derived Syntax

⟨keyword⟩	::=	. . . | `define`
⟨body⟩	::=	⟨internal-defines⟩⟨expression⟩$^+$
⟨define⟩	::=	. . . | `(define` ⟨variable⟩ ⟨expression⟩`)`
		| `(define (`⟨variable⟩ ⟨formals⟩`)` ⟨body⟩`)`
⟨internal-defines⟩	::=	⟨define⟩* (Note: all defined variables must be distinct.)
⟨let-form⟩	::=	. . . | `(`⟨let-keyword⟩ `(`⟨binding⟩*`)` ⟨body⟩`)`
⟨let-keyword⟩	::=	. . . | `let` | `let*`
⟨lambda-expression⟩	::=	. . . | `(lambda (`⟨formals⟩`)` ⟨body⟩`)` | `(lambda` ⟨variable⟩ ⟨body⟩`)`

⟨expression⟩	::=	. . . | ⟨cond⟩ | ⟨and⟩ | ⟨or⟩ |
⟨keyword⟩	::=	. . . | `cond` | `else` | `and` | `or`

⟨cond⟩	::=	`(cond` ⟨clause⟩⟨clause⟩*`)`
⟨clause⟩	::=	`(`⟨test⟩ ⟨expression⟩*`)` | `(else` ⟨expression⟩$^+$`)`
⟨and⟩	::=	`(and` ⟨expression⟩*`)`
⟨or⟩	::=	`(or` ⟨expression⟩*`)`

⟨s-expression⟩	::=	⟨identifier⟩ | ⟨self-evaluating⟩ | `(`⟨s-expression⟩*`)`

A.5 Continuations

⟨procedure-constant⟩ ::= . . . | `call/cc` | `abort`