Scheme Rewriting Model
II. Side-Effects

1 Assignment Statements

Scheme's *assignment statement* has the form

(set! ⟨variable⟩ ⟨expression⟩).

Its computational behavior is explained in part by a rewrite rule:

> `(set!` x V`)` $\longrightarrow$ `(quote set!-value)`
> where V is an immediate value and the occurrence of x is bound by some definition. This definition gets replaced by `(define` x V`)`.

The value of a `set!` expression is not very interesting, since it is always just the symbol `(quote set!-value)`. What is important about an assignment is its *side-effect*: changing the binding of a defined variable.

With side-effects in the language, the *order* in which subexpressions are evaluated plays a central role. For example, if the operands of combinations are evaluated in left to right order, then evaluation of the body

```
(define x 0)
((lambda (dummy1 dummy2) x) (set! x 1) (set! x 2))
```

returns the value `2`, but if evaluation proceeds right to left, it returns the value `1`. This is strikingly different from functional Scheme, where we have seen that a primitive value of a functional expression is independent of evaluation order.

So in a substitution model for Scheme with side-effects, we will have to specify when and where to apply rewrite rules.

2 Contexts

To be able to specify *where* to do a rewriting, we need a precise way to indicate a desired subphrase of any expression. For this purpose we introduce *contexts*.

Definition 1. Let ⟨phrase⟩ be one of the phrase types appearing in the grammar of Scheme, e.g., ⟨phrase⟩ might by ⟨variable⟩, ⟨formals⟩, ⟨binding⟩. A ⟨phrase⟩-*context* is a syntactic object, C, with a single occurrence of a variable, `[ ]`, called the *hole*, such that C would parse as a Scheme body if the hole was parsed as a ⟨phrase⟩. If C is a ⟨phrase⟩-context and P is a ⟨phrase⟩, then $C[P]$ denotes the result of literally replacing—without any renaming—the hole in C by P.

Example.

The expression

```
(if (= -6 ((lambda (y) (* 2 y) ((lambda (x) (x 3)) -)))) 5 6)
```

has a subexpression which matches the left hand side of the lambda application rule at the ⟨expression⟩-contexts C_1, C_2 where

$C_1 =$ `(if (= -6 [ ]) 5 6)`,
$C_2 =$ `(if (= -6 ((lambda (y) (* 2 y) [ ]))) 5 6)`

The body

```
(define x ())
( (lambda (x y) ( (lambda (dummy) x) (set! x (cons y x)) ))
  x
  4 )
```

has two binding occurrences of `x` at the ⟨variable⟩-contexts

```
(define [ ]())
( (lambda (x y) ( (lambda (dummy) x) (set! x (cons y x)) ))
 x
 4 )
```

and

```
(define x ())
( (lambda ([ ]  y) ( (lambda (dummy) x) (set! x (cons y x)) ))
x
4)
```

One technicality of Scheme grammar which we will use later is that a Scheme body must parse as a sequence of zero or more definitions of distinct variables followed by an expression. The Scheme grammar does *not* allow a body beginning with two or more definitions to be parsed as a definition followed by a body beginning with one or more definition. So

Example.

```
(define x 1)
[ ]
```

is an ⟨expression⟩-context, but *not* a ⟨body⟩-context.

```
(list (quote [ ]) (quote a) 3)
```

is a ⟨variable⟩-context (as well as a ⟨procedure identifier⟩- and ⟨keyword⟩-context), but *not* an ⟨expression⟩-context.

When the phrase type of the hole in a ⟨phrase⟩-context, C, is unambiguous, we drop the prefix and simply call C a "context."

3 Redexes and Contracta

Applicability of the functional rewrite rules P, C, I, L and the `set!` rule, S, is determined by the expressions which appear on the left hand sides of our descriptions of the rewrite rules:

(P)	`(prvar` $v_1 \ldots v_n$`)`	where prvar is a ⟨primitive procedure id⟩ and the v_i are ⟨primitive value⟩'s.
(C)	`(if` V M N`)`	where V is an ⟨immediate value⟩.
(I)	x	where x is a variable.
(L)	`( (lambda (`$x_1 \ldots x_n$`)` B`)` $V_1 \ldots V_n$ `)`	where the x_i are variables and the V_i are ⟨immediate value⟩'s.
(S)	`(set!` x `V)`	where x is a variable and V is an ⟨immediate value⟩.

We will refer to these expressions as *redex-triggers.* Informally we have said that rewriting consists of replacing left hand sides of rules by right hand sides. For the P and C rules, the redex-triggers above really are the complete left hand sides. However, this is not the case for I, L, and S because more information about the context in which the triggers appear is needed in order to determine how the rewriting should be carried out, or even whether the rule is applicable.

For example, consider the Instantiation Rule applied to the last occurrence of the I-trigger, i.e., variable, `k-to-the` in the expression M:

```
( (lambda (n)
    (define k n)
    (define k-to-the
```

```
      (lambda (n) (if (zero? n) 1 (* k (k-to-the (- n 1))))))
    (lambda (k) (k-to-the
                  (* k k))))
  2 )
```

In order to instantiate at all, we must know that this occurrence of `k-to-the` is bound by the definition two lines above. Not only that, but in doing the instantiation, we actually have to do more than replace the occurrence of `k-to-the` by the lambda-expression to which it is bound. To avoid improper binding, we also have to rename some of the occurrences of `k`: the body M actually rewrites to

```
( (lambda (n)
    (define k n)
    (define k-to-the
      (lambda (n) (if (zero? n) 1 (* k (k-to-the (- n 1))))))
    (lambda (k_1) ((lambda (n) (if (zero? n) 1 (* k (k-to-the (- n 1)))))
                   (* k_1 k_1))))
  2 )
```

So to describe the complete left hand side of the instantiation rule to include *all* the information needed to apply the rule and all the parts of the expression which change as a result of the rewriting, we must take the left hand side to be $D[C_1[$`k-to-the`$]]$, where D is

```
(define k n)
(define k-to-the
  (lambda (n) (if (zero? n) 1 (* k (k-to-the (- n 1))))))
[],
```

C_1 is `(lambda (k) ([ ] (* k k)))`, and the right hand side is

$D[C_1'[$`(lambda (n) (if (zero? n) 1 (* k (k-to-the (- n 1)))))`$]]$

where C_1' was obtained by freshly renaming the bound variables in C_1. Moreover, the replacement of this right hand side by this left hand side occurred *at* the context

`( (lambda (n) [ ]) 2).`

Definition 2. The complete left hand side of a rule is called its *redex*, and its complete right hand side is its *contractum.*

So for rules P and C, the redex-triggers are the redexes of the rules. For instantiation we have

Definition 3. A *definition context* is an ⟨expression⟩-context which is a sequence of one or more immediate definitions followed by a hole. An *instantiation redex* is a body of the form $D[C[x]]$ for some definition context, D, variable, x, and ⟨expression⟩-context, C, such that `(define` x V`)` is one of the definitions in D, and the occurrence of x at the hole in C is free. The *contractum* of the instantiation rule with this redex is $D[C'[V]]$ where C' is the result of freshly renaming the bound variables[1] in C.

Problem 1. Give a precise definition of whether a variable, x, has *a free occurrence at the hole in a* ⟨variable⟩*-context* C. *Hint*: By structural induction on C.

There is another form of instantiation redex for the case of instantiation of a variable within a definition context in which it is defined. A similar definition can also be given for the redexes and contracta of the Lambda application rule. We omit these general definitions. In the next section we give the special case of these rules when rewriting in evaluation order.

Problem 2. Give a precise definition for the form of instantiation redex applicable for instantiating a variable within a definition context in which it is defined.

Problem 3. Give a precise definition for the redexes and contracta of the Lambda application rule.

4 Order of Evaluation

The order in which the parts of a Scheme body are supposed to be evaluated can be described informally as follows:

- If the whole body is a redex, rewrite it into the corresponding contractum.
- If the body is a conditional, do an evaluation-rewrite on its test.
- If the body is a combination, do an evaluation-rewrite on the rightmost[2] operand or operator which is not an ⟨immediate value⟩.
- If the body is an assignment, do an evaluation-rewrite on its expression part.
- If none of the above cases apply, evaluation is complete.

An *evaluation context* is a special kind of context which we will use to formally specify where to use rules in an evaluation. We define them with a grammar which corresponds directly to the cases above:

[1] Less drastic renamings would work in many cases.

[2] This specifies right-to-left order of evaluation. The official Scheme definition allows each implementation to specify its own orders for evaluation of the components of a combination.

Definition 4. An *evaluation context* is a context of the form $D[C]$, where D is a *definition context* and C is a *control context*. Control contexts are ⟨expression⟩-contexts defined by the grammar:

$$
\begin{aligned}
\langle\text{control context}\rangle ::= {} & \texttt{[ ]} \\
& \mid \texttt{(if}\ \langle\text{control context}\rangle\ \langle\text{expression}\rangle\ \langle\text{expression}\rangle\texttt{)} \\
& \mid \texttt{(}\langle\text{expression}\rangle^*\ \langle\text{control context}\rangle\ \langle\text{immediate value}\rangle^*\texttt{)} \\
& \mid \texttt{(set!}\ \langle\text{var}\rangle\ \langle\text{control context}\rangle\texttt{)}
\end{aligned}
$$

We now specify "evaluation" versions of the rewriting rules:

Definition 5. A *Primitive application evaluation rule* has a redex of the form $E[T]$ and a corresponding contractum $E[b]$, where E is an evaluation context, T is a P-trigger and b is the ⟨primitive value⟩ determined by the trigger.

A *Conditional evaluation rule* has a redex of the form $E[T]$ and a corresponding contractum $E[M]$, where E is an evaluation context, T is a C-trigger, and M is the alternative (resp. consequent) branch of T if the test is `#f` (resp. an ⟨immediate value⟩ other than `#f`).

An *Instantiation evaluation rule* has a redex of the form $D[C[x]]$ for some *control* context, C, variable, x, whose occurrence at the hole in C is free, and definition context, D, such that `(define` x V`)` is one of the definitions in D. The corresponding *contractum* is $D[C[V]]$.

A *Lambda application evaluation rule* has a redex of the form $D[C[T]]$ where C is a *control* context, D is a definition context or hole, and T is an L-trigger. Suppose T is of the form

$$\texttt{((lambda (}x_1 \ldots x_n\texttt{)}\ D_1[M]\texttt{)}\ V_1 \ldots V_n\ \texttt{)}$$

where D_1 is a definition context or hole, M is an ⟨expression⟩, $V_1 \ldots V_n$ are ⟨immediate value⟩'s, and $x_1 \ldots x_n$ are variables, for some $n \geq 0$. Let $x'_1 \ldots x'_n$ be "fresh" variables (i.e., variables which do not occur in the redex). Then the corresponding contractum is $D[D_0[D'_1[C[M']]]]$ where D_0 is

$$\texttt{(define}\ x'_1\ V_1\texttt{)} \ldots \texttt{(define}\ x'_n\ V_n\texttt{)[ ]}$$

and $D'_1[M']$ is the result of first freshly renaming the bound variables in $D_1[M]$ (this is to avoid improper binding of free variables in C) and then replacing free occurrences of x_i by x'_i.

A `set!` *evaluation rule* has a redex of the form $D[C[$`(set!` x V`)`$]]$ where x is a variable, V is an ⟨immediate value⟩, C is a control context, and D is a definition context which includes a definition of x. The corresponding contractum is $D'[C[$`(quote set!-value)`$]]$ where D' is the same as D except that the definition of x is replaced by `(define` x V`)`.

A *Garbage collection evaluation rule* is $D[M] \longrightarrow D'[M]$ where D is a definition context and D' is the result of deleting definitions which are garbage in $D[M]$.

The evaluation rewrite rules other than garbage collection rules are called *active* rules.

A body B_1 *evaluates in one step* to body B_2, in symbols $B_1 \underset{\text{eval}}{\longrightarrow} B_2$, iff $B_1 \longrightarrow B_2$ is an active evaluation rewrite rule. B_1 *evaluates through* body B_2, written $B_1 \underset{\text{eval}}{\longrightarrow}^* B_2$, iff B_1 can be rewritten to B_2 by a series of zero or more rewritings each of which is one evaluation step *or* an application of the Garbage Collection rule.

Lemma 1. (**Determinism**) For any scheme body, B, there is at most one evaluation context, E, and active trigger, T, such that B is of the form $E[T]$. Hence, for any Scheme body, B, there is (up to fresh renaming) at most one body B' such that $B \underset{\text{eval}}{\longrightarrow} B'$.

Problem 4. Prove Lemma 1.

Definition 6. A Scheme *value* is a body of the form:

$$\langle \text{immediate define} \rangle^* \ \langle \text{immediate value} \rangle \,.$$

Lemma 2. If V is a value, then V does not evaluate in one step to anything.

Problem 5. Prove Lemma 2.

One of the virtues of a language like Scheme with automatic "garbage-collection" is that a programmer can ignore most aspects of memory allocation. This is reflected in our substitution model by the fact that, even with side-effects, garbage collection rules may be applied at any time:

Lemma 3. Rule G commutes with the active rules. See Problem Set 2, problem 4.

The Lemmas above directly imply:

Theorem 1. (**Uniqueness**) For every Scheme body, B, there is (up to freshly renaming bound variables) at most one garbage-free value, V, such that $B \underset{\text{eval}}{\longrightarrow}^* V$.

Definition 7. For a Scheme body, B, the unique garbage-free value V, if any, such that $B \underset{\text{eval}}{\longrightarrow}^* V$ is called the *evaluation-value* of B, written ev-val(B). B is said to *halt* iff $valof B$ is defined.

Bodies may not have values because they successively evaluate in one step forever, or because they get stuck after some number of evaluation steps at a *non*value expression, which does not evaluate further. Such "stuck" expressions, correspond to errors which would be reported by real Scheme.

Example. The body

```
(define f (lambda (x) (+ x (f x))))
(f 1)
```

evaluates through an infinite sequence of bodies of the form

```
(define f (lambda (x) (+ x (f x))))
(+ 1 (+ 1 (+ 1 ... (+ 1 (f 1)))))
```

and never reaches a value; evaluation in real Scheme would eventually cause a "stack overflow" error.

The body

```
(define f (lambda (x) (f x)))
(f 1)
```

evaluates to itself in three steps and a garbage collection; in real Scheme it simply keeps evaluating until it gets interrupted.

The expression

```
(if (< 3 4) f 7)
```

gets stuck at the variable `f` after two evaluation steps; evaluation in real Scheme causes an "`unbound variable f`" error.

The expression

```
((lambda (x y) (+ x y)) (* 3 4))
```

gets stuck after one step; real Scheme reports a "`compound procedure called with 1 argument; it requires exactly 2 arguments`" error.

Another important general property of evaluation is that in the functional case, evaluation rewriting does as well at getting to some value as any possible way of rewriting:

Theorem 2. Value-safety. For any body, B, in the functional kernel of Scheme, the following are equivalent

- there is a way to rewrite B into a value by successive use of the functional Scheme rewrite rules,
- B evaluates through some value,

- B halts.

Proving this theorem requires a more elaborate development of facts about term rewriting than we can undertake in 6.044.

Note that our definition of evaluation contexts implies that combinations get evaluated in right-to-left order. This was an arbitrary choice; a left-to-right or other evaluation order for combinations could have been specified. As we saw above, the values of particular expressions will not be the same using another order than ours, but the general properties of evaluation such as uniqueness and value-safety remain the same.

Problem 6. Explain how to modify the definition of evaluation context to reflect left-to-right order for the operator and operands of combinations.

5 Non-immediate Definitions

With `set!` in the kernel, we can handle non-immediate definitions as sugar. For example, we can translate the body

`(define x` $\langle\text{expression}_1\rangle$`)`
$\langle\text{expression}_2\rangle$

into

`(define x undefined-flag)`
`(begin (set! x` $\langle\text{expression}_1\rangle$`)` $\langle\text{expression}_2\rangle$`)`

where undefined-flag is some designated[3] ⟨immediate value⟩, and

`(begin` $\langle\text{expression}_0\rangle$ $\langle\text{expression}_1\rangle$`)`

is itself sugar for

`((lambda (fresh)` $\langle\text{expression}_1\rangle$`)` $\langle\text{expression}_0\rangle$`)`.

6 Other Side-effects

In this section we briefly indicate how our substitution model could be extended to include list mutation and input-output.

So far we have modeled Scheme solely by rewriting Scheme bodies. That is, if B_1 parses as a Scheme body according to our given grammar for Scheme, and $B_1 \xrightarrow[\text{eval}]{} B_2$, then B_2 also parses as a body.

[3] In real Scheme, an attempt to evaluate undefined-flag causes an error.

We were able to stick to Scheme bodies to reflect the side-effects of assigments because we could represent the results of such side-effects by changes in the ⟨define⟩'s in a body. It would be nice to model the procedures `set-car!` or `set-cdr!` which cause side-effects on lists in the same way. However, so far no satisfactory way has been found to model list side-effects while staying strictly within Scheme grammar. So we extend Scheme with new self-evaluating tokens `#cell-0`, `#cell-1`, ... , called *cell-values*. The idea is that cell-values correspond to the cons-cells in the standard internal representation of list structure maintained by Scheme implementations. Cell-values get created by new rewriting rules for list expressions, and they can be *declared* within a new `shared` special form. For example, the value of

```
(define a (list 1 2 3))
(list a (cdr a))
```

would print out in real Scheme as

```
((1 2 3) (2 3))
```

and rewrite in our Scheme TRM to the immediate value:

```
(cons
 (cons 1 (cons 2 (cons 3 ())))
 (cons (cons 2 (cons 3 ()))
       ())) .
```

But in the model with the sharing extension, this value will rewrite to a list representation which reflects the fact that the two occurrences of `(cons 2 (cons 3 ()))` above correspond to the *same* list:

```
(shared ((#cell-0 (cons 2 (cons 3 ()))))
 (cons
  (cons 1 #cell-0)
  (cons #cell-0
       ())) .
```

The Donkey interpreter in the DrScheme system at Rice University, available on the WWW at

`http://www.cs.rice.edu/CS/PLT/packages/drscheme/`

supports `shared` declarations.

Scheme's `read` and `display` procedures can be modeled with rewriting by explicitly defining variables `read-list` and `display-list` to be lists of values to be read and displayed. Then, the procedure variable `read` can be regarded as sugar for

```
(lambda ()
 (let ((value (car read-list)))
  (set! read-list (cdr read-list))
  value))
```

and `display` as sugar for

```
(lambda (output)
 (set! display-list (cons output display-list)))
```

A Grammar for Scheme with Side-Effects

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

A.1 Kernel Syntax

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

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

⟨assignment⟩ ::= `(set!` ⟨variable⟩ ⟨expression⟩`)`

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

⟨procedure identifier⟩ ::= ⟨primitive procedure id⟩ | ⟨rule-specified procedure id⟩
⟨primitive procedure id⟩ ::= `+` | `-` | `*` | `/` | `=` | `<` | `atan` | `zero?` | . . .
⟨rule-specified procedure id⟩ ::= `not` | `boolean?` | `integer?` | `number?` | `null?` | `pair?`
| `symbol?` | `procedure?` | `car` | `cdr` | `cons` | `list`
| `apply` | `eq?`

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

⟨combination⟩ ::= `(` ⟨operator⟩ ⟨operand⟩* `)`
⟨operator⟩ ::= ⟨expression⟩
⟨operand⟩ ::= ⟨expression⟩

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

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

⟨keyword⟩ ::= `define` | `lambda` | `if` | `quote` | `set!`
⟨variable⟩ ::= identifiers which are not ⟨self-evaluating⟩ ,
⟨procedure identifier⟩'s or ⟨keyword⟩'s

A.2 Values

⟨primitive value⟩ ::= ⟨self-evaluating⟩ | ⟨symbol⟩ | `()`
⟨immediate value⟩ ::= ⟨primitive value⟩ | ⟨procedure identifier⟩ | ⟨lambda expression⟩
| `(cons` ⟨immediate value⟩ ⟨immediate value⟩`)`
⟨value⟩ ::= ⟨immediate define⟩* ⟨immediate value⟩
(Note: all defined ⟨variable⟩'s must be distinct.)

A.3 Derived Syntax

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

⟨begin⟩ ::= `(begin` ⟨expression⟩ ⟨expression⟩*`)`

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

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

⟨keyword⟩ ::= ... | `cond` | `let` | `else` | `begin`