MIT/LCS/TM-281

WHAT PRICE FOR ELIMINATING EXPRESSION SIDE-EFFECTS?

Max Hailperin

June 1985

What Price for Eliminating Expression Side-effects?[*]

Max Hailperin
Laboratory for Computer Science,
Massachusetts Institute of Technology

Abstract

Separating a programming language into side-effect-free expressions and effect-only statements should make the language more amenable to axiomatization, as well as providing benefits for style, pedagogy, and implementation efficiency (particularly in parallel-computing environments). This paper shows that such a division does not come at an unreasonable cost in programming convenience. First a dialect of Lisp is defined, in which a distinction is made between statements, which may have side-effects, and expressions, which may not. Next, a representative collection of examples from Abelson and Sussman's Structure and Interpretation of Computer Programs is coded in this dialect of Lisp. Most of the examples divide neatly into functional and imperative portions, and a few relatively clean transformations prove sufficient for the more stubborn cases.

CR Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.4 [**Programming Techniques**]: Sequential Programming; D.1.m [**Programming Techniques**]: Miscellaneous; D.3.1 [**Programming Languages**]: Formal Definitions and Theory -- semantics, syntax; D.3.2 [**Programming Languages**]: Language Classifications -- applicative languages, Lisp; D.3.3 [**Programming Languages**]: Language Constructs -- procedures, functions and subroutines; D.3.m [**Programming Languages**]: Miscellaneous; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs -- assertions, logics of programs, pre- and post-conditions; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages -- operational semantics, denotational semantics; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs -- control primitives, functional constructs, program and recursion schemes.

General Terms

Experimentation, Human Factors, Languages, Design, Theory, Verification.

Additional Key Words and Phrases

Side-effects, Programming Style.

---

[*] © Max Hailperin, 1985. This paper is a revised version of the author's bachelor's thesis of the same title. It owes much to his thesis supervisor, Albert R. Meyer, who is supported in part by NSF grant MCS80-10707.

Table of Contents

Introduction

This paper provides representative empirical evidence as to the cost in lost programming convenience to be paid for ridding expressions of side-effects. First a dialect of Lisp is defined, in which a distinction is made between statements and expressions, only the former being allowed to have side-effects. Next, a representative collection of examples and exercises from Abelson and Sussman's _Structure and Interpretation of Computer Programs_[1] is coded in this dialect of Lisp. Certain programming styles are found to be negligibly affected, for example stream processing, while others are found to show slightly greater impact, for example object-oriented programming. Even the latter, though, can be handled by a relatively clean transformation.

This paper should be read in conjunction with the above-mentioned book, as no attempt is made to provide adequate explanations of the intent or context of the examples.

The primary motivation for separating a language into value-returning and effect-causing parts is that side-effect free expressions can be easily axiomatized. Eliminating all mingling of effects with values facilitates the axiomatization of the entire language. This approach has previously been taken with Algol-like languages by Meyer and others.[2] The essential difference between Algol-like languages and Lisps is that the former employ the local storage discipline (allocation from a stack rather than a heap).

Secondary motivations for the separation are that it can improve the understandability of code, provide a useful pedagogical aid, and help compilers for parallel architectures generate efficient code.

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman, _Structure and Interpretation of Computer Programs_, Cambridge: MIT Press, 1985.

[2] B. A. Trakhtenbrot, Joseph Y. Halpern, and Albert R. Meyer. "From Denotational to Operational and Axiomatic Semantics for Algol-like Languages: An Overview," MIT/LCS/TM-246, Cambridge: MIT, 1983.

The motivations listed above explain the decision to depart from conventional Lisp; the contrast between the language discipline investigated here and pure functional programming is an attempt to apply "Einstein's razor": "Everything should be made as simple as possible, but not simpler."

A Lisp Variant

The dialect of Lisp used for this paper, referred to henceforth as "Scheme-ES," is intended to resemble Scheme as far as possible, aside from the separation of statements from expressions. Other major differences are that argument passing is call-by-name rather than call-by-value (so that, for example, cons-stream[3] can be a normal function), that environments and stores are treated separately, and that identifiers and symbols are considered independent (so that, for example, EVAL does not make any sense).

Scheme-ES is defined in levels as a kernel language, primitives, sugarings, and lastly an interpreter user interface. The kernel language is not strictly speaking minimal, but does contain a fairly sparse set of constructs. The sugarings are intended to make the language more reasonable to express programs in statically, while the interpreter user interface exists solely to facilitate the dynamic creation of programs.

Kernel language

The syntax of the kernel language is given in Figure 1, and the informal semantics of each construct in Figure 2.

[3] Abelson and Sussman, p. 261.

Figure 1: Syntax of the Scheme-ES Kernel Language

lexical categories are in bold:

(	begin list
)	end list
id	identifier
atom	atom: number, string, symbol, primitive function or procedure
func	reserved word
proc	" "
if	" "
local	" "
cont	" "
letrec	" "
new	" "
set	" "
sequence	" "
call	" "

metacharacters are in roman:

::=	is defined as
{}	zero or more repetitions of
|	or

and nonterminals are in italics:

expr	expression
stmt	statement

expr ::= **atom** | **id** | **(***expr* {*expr*}**)** | **(func (**{**id**}**)** *expr***)** | **(proc** *stmt***)** | **(if** *expr* *expr* *expr***)** | **(local** *stmt* *expr***)** | **(cont** *expr***)** | **(letrec (**{**(id** *expr***)**}**)** *expr***)**

stmt ::= **(call** *expr***)** | **(new (**{**id**}**)** *stmt***)** | **(set** *expr* *expr***)** | **(sequence** {*stmt*}**)**

Figure 2: Semantics of the Scheme-ES Kernel Language

atom An atom evaluates to itself.

id An identifier evaluates to whatever it is bound to at its most closely surrounding binding. Calling is by-name, so if this binding is by a function application, rather than a **let** or **letrec**, the bound expression must be evaluated in the calling environment.

(*expr* {*expr*}**)**
The first expression is evaluated, and must evaluate to a function object; it is then applied by-name to the remaining expressions. This is accomplished by evaluating its body expression in its definition environment, augmented by binding the formals to the unevaluated arguments.

(func ({**id**}) _expr_)
A function definition evaluates to a function object with the identifiers as its formals, the expression as its body expression, and the current environment as its definition environment.

(proc _stmt_)
A procedure definition evaluates to a procedure object with the statement as its body statement and the current environment as its definition environment.

(if _expr_ _expr_ _expr_)
The first expression of a conditional is evaluated; if it evaluates to the special symbol NIL, the third expression is evaluated to provide the value of the conditional expression, otherwise, the second expression is.

(local _stmt_ _expr_)
The expression is evaluated in a private copy of the store, in which the statement has first been executed. The resulting value is the value of the **local** expression. This definition is somewhat faulty in the presence of primitive procedures, such as PRINT, that have side-effects other than on the store. Therefore, it is illegal to call such procedures from within a **local**.

(cont _expr_)
The expression is evaluated, and must have a location object as its value. The value of the overall expression is the value associated in the store with that location object.

(letrec ({(**id** _expr_)}) _expr_)
A **letrec** binds the identifiers to the results of evaluating the corresponding expressions, and then evaluates the body expression. The scope of the bindings includes both the body expression and the expressions whose values are being bound.

(call _expr_)
A procedure call, when executed, forces the evaluation of the expression, whose value must be a procedure object. That procedure object's body statement is then executed in its definition environment.

(new ({**id**}) _stmt_)
To execute a **new**, extend the store to include as many new locations as there are identifiers, and bind the identifiers to the new locations. Then execute the statement. Note that while the scope of the identifiers is the body statement, the extent of the locations is semi-infinite.

(set _expr_ _expr_)
Execution of a **set** forces evaluation of both expressions. The first expression's value must be a location, and the second expression's value is stored into that location.

(sequence {_stmt_})
Executing a **sequence** executes the statements in order.

Primitives

Only three aspects of the primitive functions and procedures warrant discussion:

- the handling of CONS, CAR, CDR, SET-CAR!, and SET-CDR!
- the definition of EQ?
- the handling of ERROR.

There are both immutable-pair and mutable-pair primitive constructors. The function of two arguments CONS constructs immutable pairs, while the procedure of three arguments (the last of which is a location into which to store the result) MUTABLE-CONS constructs mutable pairs. The functions CAR and CDR can be applied to either form of pair. The procedures SET-CAR! and SET-CDR!, on the other hand, may only be used with cons-cells produced by MUTABLE-CONS. Note that mutable-structure constructors must be procedures, as they have the side-effect of allocating storage. Immutable-structure constructors, by contrast, do not affect the store (their implementation may consume memory, for example to build closures, but this memory is not part of the explicit store (that which is accessed by **cont**) and its allocation is not considered a side-effect).

EQ? is defined in Scheme (and in other Lisps) as being true of two objects if and only if they share the same representation. As a consequence, whether it is true of two particular objects may be implementation dependent. This definition clearly has no place in a language such as Scheme-ES, which is designed for expositional purposes. Instead, something more like Common Lisp's EQL is in order; "EQL tells whether two objects are conceptually the same, whereas EQ tells whether two objects are implementationally identical."[4]

[4]Guy L. Steele Jr., Common LISP: The Language, Burlington: Digital Press, 1984, p. 78.

Thus we define Scheme-ES's EQ? operator to be true of

- two primitive objects (symbols, numbers, strings, locations, primitive functions, or primitive procedures) if they are "the same" in the obvious, visual sense,
- two pairs if they are indistinguishable; mutable cons-cells must have originated from the same **call** to MUTABLE-CONS, but immutable pairs must merely have EQ? cars and cdrs.

It is illegal to test whether a compound function or procedure is EQ? to anything, including itself, as there is no reasonable, implementation independent, decidable definition for functional equality.

In order not to divert too much attention to error handling, which is really a side issue, ERROR will be left as a function for the purpose of this paper. In order to bring this into the Scheme-ES framework, it is defined as returning an error object encapsulating the given arguments, where error objects are treated as follows:

- all primitive functions return an error object if given one
- if the test in an **if** evaluates to an error object, the value of the conditional expression is that error object
- if the function of an application evaluates to an error object, the value of the application is that error object
- the **cont**ents of an error object is that error object
- if an error object is **call**ed, the text is printed and the program stops
- if either the location or the value in a **set** is an error object, the text is printed and the program stops.

This version of error handling is imperfect, but is good enough for most purposes, and avoids wasting too much attention on the issue.

Sugarings

The sugarings fall into five categories:

- enriching statements by sugars for **(call** expr**)**
- allowing procedures to have arguments
- facilitating the entry of s-expression data
- allowing **cond** as a short-hand for nested **ifs** and **let** as a non-recursive **letrec**
- allowing multiple statements in **proc**, **local**, and **new** as implicit **sequences.**

There is a rather sparse set of statements in the kernel language. However, **letrec** and **if** can be extended to statements as well as expressions by treating them (polymorphically) as sugars for **(call** expr**)**; for example:
(if expr stmt stmt**)** ==> **(call (if** expr **(proc** stmt**) (proc** stmt**))).**

Procedures can be allowed to have arguments by the trio of sugars:
(paramproc (id {**id**}**)** {stmt}**)** ==> **(func (id** {**id**}**) (proc** {stmt}**))**
(paramproc () {stmt}**)** ==> **(proc** {stmt}**)**
(call expr expr {expr}**)** ==> **(call (**expr expr {expr}**)).**

S-expressions may be entered using the following sugaring:
(quote (s-expr **.** s-expr**))** ==> **(**CONS **(quote** s-expr**) (quote** s-expr**))**
(quote atom) ==> **atom.**
The first case is merely a convenient shorthand for the primitive CONS, but the second case may be the only way to enter a symbol if the interpreter uses the same lexical conventions for symbols and identifiers (as is traditional).

Interpreter user interface

In order to allow the incremental construction and testing of programs, the interpreter allows the user to type in three kinds of forms:

- interpreter commands
- Scheme-ES statements
- Scheme-ES expressions

The interpreter commands include such housekeeping commands as **load**, and also two special commands, **define** and **global**. **(define id** expr**)** is a command to associate the identifier with the expression in a database of definitions. Similarly, **(global id)** is a command to enter the identifier into a database of globals. The purpose of these interpreter databases will become clear below. Additionally, **(define (id** {**id**}**)** expr**)** is a shorthand for **(define id (func (**{**id**}**)** expr**))**, and **(define (id** {**id**}**)** {stmt}**)** is a shorthand for **(define id (paramproc (**{**id**}**)** {stmt}**))**.

When the user enters a Scheme-ES statement, the interpreter wraps it first in a **letrec** consisting of all of the entries in the definitions database, and then around that wraps it in a **new** which lists all of the identifiers in the globals database. The interpreter then executes the resulting statement in an empty environment (primitives are treated as atoms, not as identifiers globally bound to the primitive) and an empty store.

Lastly, if the user enters a Scheme-ES expression, the interpreter first turns it into a statement by making it the argument to a call to the primitive PRINT procedure, and then proceeds as above.

Types

Certain aspects of this language definition are clearer when viewed in terms of the types involved. Most notably, the definition of **local** above seems rather complex and *ad hoc*, but when viewed in terms of types it proves to be a very primitive operation.

There are primitive types **number**, **symbol**, **string**, and **loc**; and two methods of combining types: cartesian product ($\times$) and function mapping ($\rightarrow$). We will call the universal type **any**, and abbreviate **loc** $\rightarrow$ **any** as **store**.

An expression whose value is of type α is of type **store** $\rightarrow \alpha$, and statements are of type **store** $\rightarrow$ **store**. Functions take expressions as their arguments and may depend on the store, so +, for example, is of type ((**store** → **number**) × (**store** → **number**)) → **store** → **number.** A procedure object is of the same type as a statement, so a procedure-object-valued expression, such as **(if (cont** x) **(proc ...) (proc ...))** is of type **store** → **store** → **store.**

Where does this help? It helps explain **local.** Viewed in terms of types, **local** is as primitive as **sequence:** they are both just functional composition. **Local** composes a function of type **store** → α with one of type **store** → **store** to get one of type **store** → α. **Sequence,** on the other hand, composes a function of type **store** → **store** with one of type **store** → **store** to get one of type **store** → **store.**

Operational semantics

A simple interpreter for Scheme-ES, written in Scheme, is included as Appendix A. This provides a more precise definition of the semantics of Scheme-ES. All the examples have been run using this interpreter.

Building Abstractions with Procedures

The material covered in chapter one of Abelson and Sussman, "Building Abstractions with Procedures,"[5] translates directly into Scheme-ES, with the exception of the Monte-Carlo tests for primality. This includes straightforward function definition, and also the use of functions as both arguments to and values of higher order functions. Figure 3 illustrates examples from chapter one.

Figure 3: Examples from Chapter One

Straightforward function definition[6]

```
(define (square-root x)
  (sqrt-iter 1 x))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
```

[5] Abelson and Sussman, pp. 1-70.
[6] Abelson and Sussman, pp. 21-22.

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) .001))

(define (improve guess x)
  (average guess (/ x guess)))

(define (average x y)
  (/ (+ x y) 2))

(define (square x)
  (* x x))
```

<u>Functions as parameters</u>[7]

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))

(define (integral f a b dx)
  (* (sum f
          (+ a (/ dx 2))
          (func (x) (+ x dx))
          b)
     dx))
```

<u>Functions as returned values</u>[8]

```
(define (deriv f dx)
  (func (x)
    (/ (- (f (+ x dx)) (f x))
       dx)))

(define (newton f guess)
  (if (good-enough? guess f)
      guess
      (newton f (improve guess f))))

(define (good-enough? guess f)
  (< (abs (f guess)) .001))

(define (improve guess f)
  (- guess (/ (f guess)
              ((deriv f .001) guess))))
```

---

[7] Abelson and Sussman, pp. 54-55.
[8] Abelson and Sussman, pp. 68-69.

<u>The troublesome case: Monte-Carlo primality testing</u>[9]

```
(define (expmod b e m)
  (cond ((= e 0) 1)
        ((even? e)
         (remainder (square (expmod b (/ e 2) m))
                    m))
        (else
         (remainder (* b (expmod b (- e 1) m))
                    m))))

(define (square x)
  (* x x))

(define (fermat-test n result)
  (new (rand-loc)
    (call random (- n 2) rand-loc)
    (let ((a (+ 2 (cont rand-loc))))
      (set result (= (expmod a n n) a)))))

(define (fast-prime? n times result)
  (if (= times 0)
      (set result t)
      (new (test-result)
        (call fermat-test n test-result)
        (if (cont test-result)
            (call fast-prime? n (- times 1) result)
            (set result nil)))))
```

Note that RANDOM has a side-effect, namely changing its own internal state, so it, FERMAT-TEST, and FAST-PRIME? all have to be procedures, rather than functions, and return their results by storing them into specified locations.

The major problem with this approach is that it eliminates some of the black-boxness of FAST-PRIME? -- it would be nice if FAST-PRIME?s callers had no need to know that it used Monte-Carlo methods. One approach to restoring its modularity, based on the observation that only the values of RANDOM <u>within</u> each call to FAST-PRIME? need be independent, is to limit the extent of the side-effects with the **local** construct. Unfortunately, this requires using a global variable, as in Figure 4.

[9] Abelson and Sussman, pp. 47-48.

Figure 4: An Approach to Monte-Carlo Functions

```
(global *local-result*)
(define (fast-prime-func? n times)
  (local (new (result)
           (call fast-prime? n times result)
           (set *local-result* (cont result)))
         (cont *local-result*)))
```

The necessity of a global variable is ameliorated by the fact that *LOCAL-RESULT* can be shared among any number of such cases, provided they follow the same discipline illustrated in Figure 4: *LOCAL-RESULT* is only set as the last statement of the **local**, and the expression of the **local** is simply **(cont** *LOCAL-RESU1T***)**. These two restrictions avoid trouble in recursive cases.

Building Abstractions with Data

The techniques of chapter two, "Building Abstractions with Data,"[10] should be amenable to translation into Scheme-ES, as mutation is not introduced until chapter three. This is indeed true of data abstraction, manifest types, and message passing; data-directed programming, on the other hand, foreshadows mutation to a limited extent in its use of PUT and GET.

If the table maintained by PUT and GET is allowed to be dynamically modified in the course of execution, then we have entered the realm of mutation, the topic of the next chapter. In this chapter, however, Abelson and Sussman use PUT only in the very limited context of top-level immediate commands used to an establish a static table. For this, the PUTs need only be grouped together into a definition for GET. This chapter intentionally tries to gloss over the PUT and GET issue, saying "For now, we can assume that PUT and GET are primitive operators included in our language."[11] Thus, it is

[10] Abelson and Sussman, pp. 71-166.
[11] Abelson and Sussman, p. 138.

closest to the spirit of this chapter to "cop out" and suggest that PUT be added as an additional interpreter user interface command, like **define** and **global**, that causes the automatically generated **letrec** to include an appropriate definition for GET.

Examples from chapter two, including the "cop out" version of data-directed programming suggested above, are shown in Figure 5.

Figure 5: Examples from Chapter Two

Data abstraction[12]

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y))))

(define (-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y))))

(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))

(define (/rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))

(define (=rat x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))

(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))

(define (numer x)
  (car x))

(define (denom x)
  (cdr x))
```

[12]Abelson and Sussman, pp. 76-79.

```
(define (print-rat x)
  (call newline)
  (call princ (numer x))
  (call princ "/")
  (call princ (denom x))
  (call princ " "))
```

<u>Manifest types</u>[13]

```
(define (attach-type type contents)
  (cons type contents))

(define (type datum)
  (if (not (atom? datum))
      (car datum)
      (error "Bad typed datum -- TYPE" datum)))

(define (contents datum)
  (if (not (atom? datum))
      (cdr datum)
      (error "Bad typed datum -- CONTENTS" datum)))

(define (rectangular? z)
  (eq? (type z) 'rectangular))

(define (polar? z)
  (eq? (type z) 'polar))

(define (make-rectangular x y)
  (attach-type 'rectangular (cons x y)))

(define (make-polar r a)
  (attach-type 'polar (cons r a)))

(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))))

(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))))

(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))))
```

[13]Abelson and Sussman, pp. 133-135.

```
(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))))

(define (real-part-rectangular z)
  (car z))

(define (imag-part-rectangular z)
  (cdr z))

(define (magnitude-rectangular z)
  (sqrt (+ (square (car z))
           (square (cdr z)))))

(define (angle-rectangular z)
  (atan (cdr z) (car z)))

(define (real-part-polar z)
  (* (car z) (cos (cdr z))))

(define (imag-part-polar z)
  (* (car z) (sin (cdr z))))

(define (magnitude-polar z)
  (car z))

(define (angle-polar z)
  (cdr z))

(define (square x)
  (* x x))
```

Message passing[14]

```
(define (make-rectangular x y)
  (func (m)
    (cond ((eq? m 'real-part) x)
          ((eq? m 'imag-part) y)
          ((eq? m 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? m 'angle) (atan y x))
          (else
           (error "Unknown op -- MAKE-RECTANGULAR" m)))))

(define (operate op obj) (obj op))

(define (square x) (* x x))
```

[14] Abelson and Sussman, pp. 141.

The troublesome case: data-directed programming[15]

```
(put 'rectangular 'real-part real-part-rectangular)
(put 'rectangular 'imag-part imag-part-rectangular)
(put 'rectangular 'magnitude magnitude-rectangular)
(put 'rectangular 'angle angle-rectangular)

(put 'polar 'real-part real-part-polar)
(put 'polar 'imag-part imag-part-polar)
(put 'polar 'magnitude magnitude-polar)
(put 'polar 'angle angle-polar)

(define (operate op obj)
  (let ((function (get (type obj) op)))
    (if (not (null? function))
        (function (contents obj))
        (error "Operator undefined for this type -- OPERATE"
               (list op obj)))))

(define (real-part obj) (operate 'real-part obj))
(define (imag-part obj) (operate 'imag-part obj))
(define (magnitude obj) (operate 'magnitude obj))
(define (angle obj) (operate 'angle obj))
```

Modularity, Objects, and State

In chapter three, "Modularity, Objects, and State,"[16] Abelson and Sussman introduce the concept of change, making Scheme-ES's separation of side-effects from expression evaluation an issue for the first time (aside from tangentially in the Monte-Carlo primality tests of chapter one and the data-directed programs of chapter two). We will see that stream processing (as expected) is not adversely affected, and (more surprisingly) that even object-oriented programming is only mildly affected.

Assignments and local state

Abelson and Sussman start by using a bank account as an example of the use of local state variables.[17] In their version, making a withdrawal results in the remaining balance being returned, or an "Insufficient funds" message. This means that WITHDRAW has both an effect and a value -- an impossibility in

[15] Abelson and Sussman, pp. 138.
[16] Abelson and Sussman, pp. 167-292.
[17] Abelson and Sussman, p. 169 ff.

Scheme-ES. One possible solution is to store the value into a location passed as an additional argument, as in Figure 6.

Another approach is to change the specifications of the problem to fit the action/value dichotomy of Scheme-ES. This could be done by separately providing a procedure and a function. The procedure would only make the withdrawal, if possible; the function would only return the account balance.

Figure 6: Procedures with Local State[18]

```
(define (make-withdraw starting-balance result)
  (new (balance)
    (set balance starting-balance)
    (set result (paramproc (amount result)
                  (if (>= (cont balance) amount)
                      (sequence (set balance (- (cont balance) amount))
                                (set result (cont balance)))
                      (set result "Insufficient funds"))))))
```

<u>Example of use</u>

```
(new (W1 W2 result)
  (call make-withdraw 100 W1)
  (call make-withdraw 100 W2)
  (call (cont W1) 50 result)
  (call print (cont result))
  (call (cont W2) 70 result)
  (call print (cont result))
  (call (cont W2) 40 result)
  (call print (cont result))
  (call (cont W1) 40 result)
  (call print (cont result)))
```

<u>Result of example</u>

```
50
30
"Insufficient funds"
10
```

<u>Modeling with mutable data</u>

The examples of mutable data given in this chapter all respect the action/value dichotomy, except for creation, so they can be easily converted into Scheme-ES, as illustrated below for queues. This is not a coincidence:

[18] Abelson and Sussman, p. 172.

it is good style to separate the operations of an abstract data type into constructors, selectors, and mutators. The selectors are pure functions, and the mutators are pure procedures. Only the constructors pose a problem: they have a value (the constructed mutable data object), but they also have an effect (the allocation of storage). This must typically be worked around by storing the result, as in the queue example of Figure 7, below.

Figure 7: A Mutable Data Type[19]

```
(define (front-ptr queue) (car queue))

(define (rear-ptr queue) (cdr queue))

(define (set-front-ptr! queue item) (call set-car! queue item))

(define (set-rear-ptr! queue item) (call set-cdr! queue item))

(define (empty-queue? queue) (null? (front-ptr queue)))

(define (make-queue result) (call mutable-cons '() '() result))

(define (front queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))

(define (insert-queue! queue item)
  (new (new-pair)
    (call mutable-cons item nil new-pair)
    (cond ((empty-queue? queue)
           (call set-front-ptr! queue (cont new-pair))
           (call set-rear-ptr! queue (cont new-pair)))
          (else
           (call set-cdr! (rear-ptr queue) (cont new-pair))
           (call set-rear-ptr! queue (cont new-pair))))))

(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (call error "Delete called with an empty queue" queue))
        (else
         (call set-front-ptr! queue (cdr (front-ptr queue))))))
```

---

[19] Abelson and Sussman, pp. 210-212.

The composition of constructors

The approach described above, storing results, works ok in simple cases, but it lacks the ability for composition that expression evaluation provides. The advantage of functional composition is that it allows the structure of a program that constructs hierarchical data to parallel the structure of the data. A solution to this problem is illustrated below by the simple example of mutable cons-cells; a more sophisticated example from Abelson and Sussman (an object-oriented implementation of constraint propagation) is deferred to Appendix B.

The primitive Scheme-ES constructor for mutable cons-cells is a procedure, so the only way to build mutable trees with it is by explicitly allocating temporaries for each node in the tree and building it step by step. Figure 8 illustrates this with a procedure, make-tree, which makes a complete binary tree of depth two.

Figure 8: Building a Mutable Tree Step by Step

```
(define (make-tree caar-leaf cdar-leaf cadr-leaf cddr-leaf result)
  (new (car-node cdr-node)
    (call mutable-cons caar-leaf cdar-leaf car-node)
    (call mutable-cons cadr-leaf cddr-leaf cdr-node)
    (call mutable-cons (cont car-node) (cont cdr-node) result)))
```

This worrying about step-by-step computation and temporaries for intermediate results is clearly more reminiscent of an assembly language then of a higher-level language. We should be able to rewrite make-tree as in Figure 9. Notice that the MUT-CONS calls in Figure 9 form a complete binary tree of depth two themselves.

Figure 9: Building a Mutable Tree by Functional Composition

```
(define (mk-tree caar-leaf cdar-leaf cadr-leaf cddr-leaf)
  (mut-cons (mut-cons caar-leaf cdar-leaf)
            (mut-cons cadr-leaf cddr-leaf)))
```

At first glance, the MK-TREE function of Figure 9 looks like it couldn't possibly be legal in Scheme-ES, as constructing a mutable object implies the allocation of storage. But there is a solution: in order to protect the separation of values from effects, we must separate composition from construction by introducing an additional level of abstraction.

If MUT-CONS (and thus MK-TREE) doesn't return a mutable cons-cell, but rather a procedure to create one and store it into a given location, then it will not have a side-effect. The allocation of storage does not occur until the returned procedure is called. Lastly, to allow the result of one MUT-CONS to be used as an argument to another one, the arguments must also be constructor procedures, rather than actual mutable cons-cells. Given this re-typing of its result and arguments, MUT-CONS can be expressed in Scheme-ES as in Figure 10.

Figure 10: A Function to Make Procedures to Construct Mutable Cons-cells

```
(define (mut-cons car-constructor cdr-constructor)
  (paramproc (result)
    (new (the-car the-cdr)
      (call car-constructor the-car)
      (call cdr-constructor the-cdr)
      (call mutable-cons (cont the-car) (cont the-cdr) result))))
```

This retyping can be viewed as a rather straightforward transformation on types. Suppose, for the sake of exposition, that the store is typed, such that to every type α there corresponds a location type $\mathbf{loc}_\alpha$, where only αs can be stored into $\mathbf{loc}_\alpha$s. Then the retyping of mut-cons is just a transformation from

$$((\mathbf{store} \to \alpha) \times (\mathbf{store} \to \beta)) \to \mathbf{store} \to \gamma$$

to

$$((\mathbf{store} \to (\mathbf{loc}_\alpha \to \mathbf{proc})) \times (\mathbf{store} \to (\mathbf{loc}_\beta \to \mathbf{proc}))) \to \mathbf{store} \to (\mathbf{loc}_\gamma \to \mathbf{proc}).$$

(Where **proc** is **store** $\rightarrow$ **store**.) In other words, we replace each type α, β, γ by the ability to store an object of that type: $\mathbf{loc}_\alpha \rightarrow \mathbf{proc}$, for example.[20]

The semantics of the retyped function, f, can be defined in terms of the originally typed function, f', which is the "functional essence" of f, and a procedure (of type **store** $\rightarrow$ **store**) f'', which is the "procedural essence" of f. To do this, we will need a semantic model for the **new** statement. **New** can be viewed as a function of type **store** $\rightarrow$ (**store** $\times$ **loc**), but for our purposes it will be more convenient to decompose it into two functions, NEW, of type **store** $\rightarrow$ **loc**, and ALLOC, of type **store** $\rightarrow$ **store**. Furthermore, in keeping with our fiction of a typed store, we will subscript NEW and ALLOC with the type of the location: NEW_α is of type **store** $\rightarrow \mathbf{loc}_\alpha$. We will use σ, τ, σ', etc. for **stores**, and ℓ_α, etc. for $\mathbf{loc}_\alpha$s, etc. ASSIGN_α will be the assignment primitive; its type is $\mathbf{loc}_\alpha \rightarrow \alpha \rightarrow \mathbf{store} \rightarrow \mathbf{store}$. Given this notation, we can define the semantics of f by f (x, y) $\sigma\ \ell_\gamma = \sigma''''''$, where:

$\ell_\alpha = \text{NEW}_\alpha\ \sigma$ (allocate locations for arguments)

$\sigma' = \text{ALLOC}_\alpha\ \sigma$

$\ell_\beta = \text{NEW}_\beta\ \sigma'$

$\sigma'' = \text{ALLOC}_\beta\ \sigma'$

$\sigma''' = x\ \sigma''\ \ell_\alpha\ \sigma''$ (call arguments with their locations)

$\sigma'''' = y\ \sigma'''\ \ell_\beta\ \sigma'''$

$\sigma''''' = f''\ \sigma''''$ (do the "procedural essence")

$\sigma'''''' = \text{ASSIGN}_\gamma\ \ell_\gamma\ (f'\ (\lambda\tau.(\tau\ \ell_\alpha),\ \lambda\tau.(\tau\ \ell_\beta))\ \sigma''''')\ \sigma'''''$
(set given location to value of "functional essence")

[20]This transformation from functions on objects to functions from constructors to constructors is not merely general enough to be used with any mutable-data-structure constructor, it is in fact general enough to serve as the basis of a mechanical translation from Scheme to Scheme-ES. The fact that such a translation exists does not mean that Scheme-ES provides no additional leverage, but rather merely that it is equally general. The advantage of code written directly in Scheme-ES is that it is clear where side-effects occur and where they don't, while the mechanical translation from Scheme mentioned above would conservatively assume that every function has a side-effect.

Given this new view of what kind of objects mut-cons should deal in, the definition of MK-TREE shown in Figure 9 is a completely correct and legal Scheme-ES function. The only detail remaining is how to form bridges between the abstracted world of procedures that make mutable cons-cells and the real world of mutable cons-cells and other objects. We need to cons arbitrary objects into our tree at the leaves, and we need ultimately to construct the actual mutable cons-cells and store the top-level one somewhere. Figure 11 shows these two bridges: MC-LEAF and MC-SET. Figure 12 then summarizes by illustrating an example usage.

Figure 11: Bridges between the Abstracted and Real Worlds

```
(define (mc-leaf object)
  (paramproc (result)
    (set result object)))

(define (mc-set location mc)
  (call mc location))
```

Figure 12: An Example Construction of a Mutable Tree

<u>Executing:</u>

```
(new (tree)
  (call mc-set tree (mk-tree (mc-leaf 'the-caar)
                             (mc-leaf 'the-cdar)
                             (mc-leaf 'the-cadr)
                             (mc-leaf 'the-cddr)))
  (call print (cdar (cont tree)))
  (call set-cdr! (car (cont tree)) 'new-cdar)
  (call print (cdar (cont tree))))
```

<u>Results in:</u>

```
THE-CDAR
NEW-CDAR
```

Streams

The above cases exhaust the various ways Abelson and Sussman use assignment and mutation to model change. The last remaining programming style, streams, is different, in that it is purely functional. Thus, there are absolutely no difficulties converting to Scheme-ES.

Metalinguistic Abstraction

The last two chapters of Structure and Interpretation of Computer Programs are concerned with the additional abstraction technique of metalinguistic abstraction, i.e. defining new languages. While this is a powerful tool, it is purely a conceptual hurdle, and not a distinguished programming style. From the point of view of gathering evidence on how various programming styles fair in Scheme-ES, chapters four and five merely provide further examples of the styles introduced in the first three chapters.

Conclusion

Ridding expressions of side-effects is not only theoretically and practically interesting, it is a viable language design option. A wide variety of programming techniques have been examined, and the elimination of expression side-effects has been shown to have only mild consequences. Most programming divides neatly into pure procedures and pure functions. There are only a limited number of borderline cases that have both effects and values:

- Monte-Carlo experiments
- inappropriately specified data-structure operations (such as WITHDRAW) that combine features of both mutators and selectors
- mutable data-structure constructors.

These borderline cases must be handled by the "hack" of storing their results into a location passed as an additional argument. While this hack is normally merely a bother, in two cases it is more serious:

- whether a routine uses Monte-Carlo methods or not cannot be completely hidden from its callers
- mutable data-structure constructors can not be composed.

The former problem can be handled by limiting the extent of the side-effects to the minimum range over which the random numbers must be independent. This generally provides adequate hiding of the Monte-Carlo nature of a computation, as in FAST-PRIME-FUNC? in Figure 4.

The latter problem can be handled by composing functions returning constructors, rather than the constructors themselves. Once the ultimate constructor has been synthesized, which will construct all the nested mutable data structures, then it can be called with the location into which to store the whole mess. This technique is illustrated above by the mutable-cons example of Figures 9-11.

References

Harold Abelson and Gerald Jay Sussman with Julie Sussman, _Structure and Interpretation of Computer Programs_, Cambridge: MIT Press, 1985.

Guy L. Steele Jr., _Common LISP: The Language_, Burlington: Digital Press, 1984, p. 78.

B. A. Trakhtenbrot, Joseph Y. Halpern, and Albert R. Meyer. "From Denotational to Operational and Axiomatic Semantics for Algol-like Languages: An Overview," MIT/LCS/TM-246, Cambridge: MIT, 1983.

Appendix A: A Simple Scheme-ES Interpreter

```
; WARNING: The division of Scheme-ES between kernel and sugars has shifted
;          somewhat since this interpreter was written.

; ******************************************************************************
; These functions constitute the kernel interpreter.  Kernel-exec and
; kernel-eval are the interpreter proper.  Next appear specialized
; interpreters for each statement or expression type.  Lastly, there are a
; few supporting routines for data-directed dispatching and to implement
; the closure data-types: thunks, procedures, and functions.
; ******************************************************************************

(define (kernel-exec stmt env store)
  (dispatch (stmt-type stmt) 'exec stmt env store))

(define (kernel-eval expr env store)
  (dispatch (expr-type expr) 'eval expr env store))

(define (exec-call stmt env store)
  (let ((proc (kernel-eval (call-stmt-proc stmt) env store)))
    (cond ((primitive-proc? proc)
           (call-primitive proc store))
          ((error-object? proc)
           (error (unparse-s-expr (error-object-message proc))
                  (unparse-s-expr (error-object-irritant proc))))
          (else
           (do-for-each (lambda (stmt)
                          (kernel-exec stmt (proc-env proc) store))
                        (proc-body proc))))))
(put!-prop 'call 'exec exec-call)

(define (exec-set stmt env store)
  (let ((loc (kernel-eval (set-stmt-loc stmt) env store))
        (val (kernel-eval (set-stmt-val stmt) env store)))
    (if (error-object? loc)
        (error (unparse-s-expr (error-object-message loc))
               (unparse-s-expr (error-object-irritant loc))))
    (if (error-object? val)
        (error (unparse-s-expr (error-object-message val))
               (unparse-s-expr (error-object-irritant val))))
    (assign! loc val store)))
(put!-prop 'set 'exec exec-set)
```

```
(define (exec-new stmt env store)
  (let ((ids (new-stmt-ids stmt))
        (stmts (new-stmt-stmts stmt)))
    (let ((new-env (extend-environment ids
                                        (alloc! store
                                                (length ids))
                                        env)))
      (do-for-each (lambda (stmt)
                     (kernel-exec stmt new-env store))
                   stmts))))
(put!-prop 'new 'exec exec-new)

(define (exec-sequence stmt env store)
  (do-for-each (lambda (stmt)
                 (kernel-exec stmt env store))
               (sequence-stmt-stmts stmt)))
(put!-prop 'sequence 'exec exec-sequence)

(define (eval-atom expr env store)
  (atom-expr-atom expr))
(define atom-type (gensym 'atom-type))
(put!-prop atom-type 'eval eval-atom)

(define (eval-id expr env store)
  (let ((value (lookup-variable-value (id-expr-id expr) env)))
    (if (thunk? value)
        (kernel-eval (thunk-expr value) (thunk-env value) store)
        value)))
(define id-type (gensym 'id-type))
(put!-prop id-type 'eval eval-id)

(define (eval-application expr env store)
  (let ((func-expr (application-expr-func expr))
        (args (application-expr-args expr)))
    (let ((func (kernel-eval func-expr env store)))
      (cond ((primitive-func? func)
             (apply-primitive func args env store))
            ((error-object? func)
             func)
            (else
             (kernel-eval (func-body func)
                          (extend-environment (func-formals func)
                                              (mapcar (lambda (arg)
                                                        (make-thunk arg
                                                                    env))
                                                      args)
                                              (func-env func))
                          store))))))
(define application-type (gensym 'application-type))
(put!-prop application-type 'eval eval-application)

(define (eval-func expr env store)
  (make-func (func-expr-ids expr) (func-expr-expr expr) env))
(put!-prop 'func 'eval eval-func)
```

```
(define (eval-proc expr env store)
  (make-proc (proc-expr-stmts expr) env))
(put!-prop 'proc 'eval eval-proc)

(define (eval-if expr env store)
  (let ((test-expr (if-expr-test expr))
        (true-expr (if-expr-true expr))
        (false-expr (if-expr-false expr)))
    (let ((test (kernel-eval test-expr env store)))
      (cond ((error-object? test)
             test)
            (test
             (kernel-eval true-expr env store))
            (else
             (kernel-eval false-expr env store))))))
(put!-prop 'if 'eval eval-if)

(define (eval-local expr env store)
  (let ((stmts (local-expr-stmts expr))
        (expr (local-expr-expr expr))
        (new-store (copy-store store))
        (old-effects-flag effects-allowed?)
        (old-randomize-to randomize-to))
    (set! effects-allowed? nil)
    (do-for-each (lambda (stmt)
                   (kernel-exec stmt env new-store))
                 stmts)
    (set! randomize-to old-randomize-to)
    (set! effects-allowed? old-effects-flag)
    (kernel-eval expr env new-store)))
(put!-prop 'local 'eval eval-local)

(define (eval-cont expr env store)
  (let ((loc (kernel-eval (cont-expr-loc expr) env store)))
    (if (error-object? loc)
        loc
        (fetch loc store))))
(put!-prop 'cont 'eval eval-cont)

(define (eval-let expr env store)
  (let ((ids (mapcar let-pair-id (let-expr-pairs expr)))
        (values (mapcar (lambda (expr)
                          (kernel-eval expr env store))
                        (mapcar let-pair-expr (let-expr-pairs expr))))
        (expr (let-expr-expr expr)))
    (kernel-eval expr (extend-environment ids values env) store)))
(put!-prop 'let 'eval eval-let)
```

```
(define (eval-letrec expr env store)
  (let ((ids (mapcar letrec-pair-id (letrec-expr-pairs expr)))
        (exprs (mapcar letrec-pair-expr (let-expr-pairs expr))))
    (let ((outer-values (mapcar (lambda (expr)
                                  (make-func ids
                                             `((func ,ids
                                                   ,expr)
                                               . ,(mapcar (lambda (id)
                                                            `(,id . ,ids))
                                                          ids))
                                             env))
                                exprs)))
      (let ((outer-env (extend-environment ids outer-values env)))
        (let ((inner-values (mapcar (lambda (id)
                                      (kernel-eval `(,id . ,ids)
                                                   outer-env
                                                   store))
                                    ids)))
          (let ((inner-env (extend-environment ids
                                               inner-values
                                               outer-env)))
            (kernel-eval (letrec-expr-expr expr) inner-env store)))))))
(put!-prop 'letrec 'eval eval-letrec)

(define (dispatch type op . args)
  (let ((proc (get-prop type op)))
    (if (null? proc)
        (error "DISPATCH: operation undefined for type" (list op type))
        (apply proc args))))

(define (make-thunk expr env)
  `(thunk ,expr ,env))

(define (thunk? x)
  (if (atom? x)
      nil
      (eq? (car x) 'thunk)))

(define thunk-expr cadr)

(define thunk-env caddr)

(define (make-proc body env)
  `(compound-proc ,body ,env))

(define (compound-proc? x)
  (if (pair? x)
      (eq? (car x) 'compound-proc)
      nil))
```

```
(define (proc-body x)
  (if (atom? x)
      (error "PROC-BODY: non-procedure" x)
      (if (eq? (car x) 'compound-proc)
          (cadr x)
          (error "PROC-BODY: non-procedure" x))))

(define (proc-env x)
  (if (atom? x)
      (error "PROC-ENV: non-procedure" x)
      (if (eq? (car x) 'compound-proc)
          (caddr x)
          (error "PROC-ENV: non-procedure" x))))

(define (make-func formals body env)
  `(compound-func ,formals ,body ,env))

(define (compound-func? x)
  (if (pair? x)
      (eq? (car x) 'compound-func)
      nil))

(define (func-formals x)
  (if (atom? x)
      (error "FUNC-FORMALS: non-function" x)
      (if (eq? (car x) 'compound-func)
          (cadr x)
          (error "FUNC-FORMALS: non-function" x))))

(define (func-body x)
  (if (atom? x)
      (error "FUNC-BODY: non-function" x)
      (if (eq? (car x) 'compound-func)
          (caddr x)
          (error "FUNC-BODY: non-function" x))))

(define (func-env x)
  (if (atom? x)
      (error "FUNC-ENV: non-function" x)
      (if (eq? (car x) 'compound-func)
          (cadddr x)
          (error "FUNC-ENV: non-function" x))))

(define (do-for-each proc list)
  (if (not (null? list))
      (sequence (proc (car list))
                (do-for-each proc (cdr list)))))
```

```
; ******************************************************************************
; These functions form the interpreter itself: the read-eval-print loop
; and the interpreter user-interface commands.
; ******************************************************************************

(define (interpreter)
  (initialize)
  (r-e-p))

(define (initialize)
  (set! effects-allowed? t)
  (set! the-definitions '())
  (set! the-globals '())
  (set! the-get-cases '((else nil))))

(define (r-e-p) ; the read-eval-print loop
  (newline)
  (princ "Scheme-ES> ")
  (process-input (read))
  (r-e-p))

(define (process-input form)
  (cond ((command? form)
         (do-command form))
        ((stmt? form)
         (kernel-exec (unsugar (wrap form))
                      the-empty-environment
                      (make-store)))
        (else
         (kernel-exec (unsugar (wrap `(call print ,form)))
                      the-empty-environment
                      (make-store)))))

(define (command? form)
  (if (pair? form)
      (not (null? (get-prop (car form) 'command)))
      nil))

(define (do-command form)
  ((get-prop (car form) 'command) form))

(define the-definitions '())
(define the-globals '())
(define the-get-cases '((else nil)))

(define (wrap stmt)
  `(new ,the-globals
     (letrec ,(cons `(get (func (tag1 tag2) (cond . ,the-get-cases)))
                    the-definitions)
       ,stmt)))
```

```
(put!-prop 'global
           'command
           (lambda (form)
             (if (atom? (cdr form))
                 (error "Illegal GLOBAL syntax" form)
                 (if (not (memq (cadr form) the-globals))
                     (set! the-globals (cons (cadr form) the-globals))))))

(put!-prop 'put
           'command
           (lambda (form)
             (if (atom? (cdddr form))
                 (error "Illegal PUT syntax" form)
                 (let ((old-case (member `(and (eq? tag1 ,(cadr form))
                                               (eq? tag2 ,(caddr form)))
                                         the-get-cases)))
                   (if (null? old-case)
                       (set! the-get-cases (cons `((and (eq? tag1
                                                              ,(cadr form))
                                                         (eq? tag2
                                                              ,(caddr form)))
                                                    ,(cadddr form))
                                                  the-get-cases))
                       (set-car! (cdr old-case) (cadddr form)))))))

(put!-prop 'define
           'command
           (lambda (form)
             (let ((old-def (memq (definition-name form) the-definitions)))
               (if (null? old-def)
                   (set! the-definitions (cons `(,(definition-name form)
                                                 ,(definition-expr form))
                                               the-definitions))
                   (set-car! (cdr old-definition)
                             (definition-expr form))))))

(define (definition-name form)
  (if (atom? (cdr form))
      (error "Illegal DEFINE syntax" form)
      (if (atom? (cadr form))
          (cadr form)
          (caadr form))))

(define (definition-expr form)
  (if (atom? (cddr form))
      (error "Illegal DEFINE syntax" form)
      (if (atom? (cadr form))
          (caddr form)
          (if (stmt? (caddr form))
              `(paramproc ,(cdadr form) . ,(cddr form))
              `(func ,(cdadr form) ,(caddr form))))))
```

```
(put!-prop 'load          ; load in from a Scheme variable
           'command
           (lambda (form)
             (do-for-each process-input (eval (cadr form) (current-env)))))

(define (current-env)
  (frame-parent (frame-parent (make-environment))))


; ******************************************************************************
; These functions unsugar syntactic sugars.
; ******************************************************************************

(define (unsugar form)
  (if (stmt? form)
      (dispatch (stmt-type form) 'unsugar-stmt form)
      (dispatch (expr-type form) 'unsugar-expr form)))

(put!-prop 'call
           'unsugar-stmt
           (lambda (form)
             (let ((args (call-stmt-args form)))
               (if (null? args)
                   `(call ,(unsugar (call-stmt-proc form)))
                   `(call (,(unsugar (call-stmt-proc form))
                           . ,(mapcar unsugar args)))))))

(put!-prop 'new
           'unsugar-stmt
           (lambda (form)
             `(new ,(new-stmt-ids form)
                . ,(mapcar unsugar
                           (new-stmt-stmts form)))))

(put!-prop 'set
           'unsugar-stmt
           (lambda (form)
             `(set ,(unsugar (set-stmt-loc form))
                   ,(unsugar (set-stmt-val form)))))

(put!-prop 'sequence
           'unsugar-stmt
           (lambda (form)
             `(sequence . ,(mapcar unsugar (sequence-stmt-stmts form)))))

(put!-prop 'if
           'unsugar-stmt
           (lambda (form)
             `(call (if ,(unsugar (if-expr-test form))
                        (proc ,(unsugar (if-expr-true form)))
                        (proc ,(unsugar (if-expr-false form)))))))
```

```
(put!-prop 'let
           'unsugar-stmt
           (lambda (form)
             `(call (let ,(unsugar-let-pairs (let-expr-pairs form))
                     (proc . ,(mapcar unsugar (let-stmt-stmts form)))))))

(define (unsugar-let-pairs pairs)
  (if (null? pairs)
      '()
      `((,(let-pair-id (car pairs))
         ,(unsugar (let-pair-expr (car pairs))))
        . ,(unsugar-let-pairs (cdr pairs)))))

(put!-prop 'letrec
           'unsugar-stmt
           (lambda (form)
             `(call (letrec ,(unsugar-letrec-pairs
                              (letrec-expr-pairs form))
                     (proc . ,(mapcar unsugar
                                      (letrec-stmt-stmts form)))))))

(define (unsugar-letrec-pairs pairs)
  (if (null? pairs)
      '()
      `((,(letrec-pair-id (car pairs))
         ,(unsugar (letrec-pair-expr (car pairs))))
        . ,(unsugar-letrec-pairs (cdr pairs)))))

(put!-prop 'cond
           'unsugar-stmt
           (lambda (form)
             `(call ,(ifize-cond-stmt-clauses (cond-expr-clauses form)))))

(define (ifize-cond-stmt-clauses clauses)
  (if (null? clauses)
      '(proc)
      (if (eq? (cond-clause-test (car clauses)) 'else)
          `(proc . ,(mapcar unsugar (cond-clause-stmts (car clauses))))
          `(if ,(unsugar (cond-clause-test (car clauses)))
               (proc . ,(mapcar unsugar (cond-clause-stmts (car clauses))))
               ,(ifize-cond-stmt-clauses (cdr clauses))))))

(put!-prop atom-type
           'unsugar-expr
           (lambda (form)
             (if (quoted? form)
                 `(quote ,(parse-s-expr (quoted-expr-body form)))
                 form)))

(put!-prop id-type 'unsugar-expr (lambda (form) form))
```

```
(put!-prop application-type
           'unsugar-expr
           (lambda (form)
             `(,(unsugar (application-expr-func form))
               . ,(mapcar unsugar (application-expr-args form)))))

(put!-prop 'func
           'unsugar-expr
           (lambda (form)
             `(func ,(func-expr-ids form)
                ,(unsugar (func-expr-expr form)))))

(put!-prop 'proc
           'unsugar-expr
           (lambda (form)
             `(proc . ,(mapcar unsugar (proc-expr-stmts form)))))

(put!-prop 'paramproc
           'unsugar-expr
           (lambda (form)
             (let ((formals (paramproc-expr-ids form)))
               (if (null? formals)
                   `(proc . ,(mapcar unsugar (paramproc-expr-stmts form)))
                   `(func ,formals
                      (proc
                        . ,(mapcar unsugar
                                   (paramproc-expr-stmts form)))))))))

(put!-prop 'if
           'unsugar-expr
           (lambda (form)
             `(if ,(unsugar (if-expr-test form))
                  ,(unsugar (if-expr-true form))
                  ,(unsugar (if-expr-false form)))))

(put!-prop 'local
           'unsugar-expr
           (lambda (form)
             `(local . ,(mapcar unsugar
                                (local-expr-stmts-and-expr form)))))

(put!-prop 'cont
           'unsugar-expr
           (lambda (form)
             `(cont ,(unsugar (cont-expr-loc form)))))

(put!-prop 'let
           'unsugar-expr
           (lambda (form)
             `(let ,(unsugar-let-pairs (let-expr-pairs form))
                ,(unsugar (let-expr-expr form)))))
```

```
(put!-prop 'letrec
           'unsugar-expr
           (lambda (form)
             `(letrec ,(unsugar-letrec-pairs (letrec-expr-pairs form))
                ,(unsugar (letrec-expr-expr form)))))

(put!-prop 'cond
           'unsugar-expr
           (lambda (form)
             (ifize-cond-expr-clauses (cond-expr-clauses form))))

(define (ifize-cond-expr-clauses clauses)
  (if (null? clauses)
      nil
      (if (eq? (cond-clause-test (car clauses)) 'else)
          (unsugar (cond-clause-expr (car clauses)))
          `(if ,(unsugar (cond-clause-test (car clauses)))
               ,(unsugar (cond-clause-expr (car clauses)))
               ,(ifize-cond-expr-clauses (cdr clauses))))))


; ******************************************************************************
; These functions implement environments and stores.  Stores are trivially
; implemented as environments (with one frame per local), and environments
; are copied from Abelson and Sussman, Structure and Interpretation of
; Computer Programs, pp. 306-309.
; ******************************************************************************

(define (alloc! store n)  ; extend store by n locations and return them
  (if (= n 0)
      '()
      (let ((new-loc (gensym 'loc)))
        (define-variable! new-loc '*unassigned* store)
        (cons new-loc (alloc! store (-1+ n))))))

(define (copy-store store)
  (extend-environment '() '() store))

(define (fetch loc store)
  (lookup-variable-value loc store))

(define (assign! loc val store)
  (lookup-variable-value loc store)  ; signals error if non-location
  (define-variable! loc val store))

(define (make-store)
  (extend-environment '() '() the-empty-environment))

(define (lookup-variable-value var env)
  (let ((b (binding-in-env var env)))
    (if (found-binding? b)
        (binding-value b)
        (error "Unbound variable or non-location" var))))
```

```
(define (binding-in-env var env)
  (if (no-more-frames? env)
      no-binding
      (let ((b (binding-in-frame var (first-frame env))))
        (if (found-binding? b)
            b
            (binding-in-env var (rest-frames env))))))

(define (extend-environment variables values base-env)
  (adjoin-frame (make-frame variables values) base-env))

(define (set-variable-value! var val env)
  (let ((b (binding-in-env var env)))
    (if (found-binding? b)
        (set-binding-value! b val)
        (error "Unbound variable" var))))

(define (define-variable! var val env)
  (let ((b (binding-in-frame var (first-frame env))))
    (if (found-binding? b)
        (set-binding-value! b val)
        (set-first-frame! env
                          (adjoin-binding (make-binding var val)
                                          (first-frame env))))))

(define (first-frame env) (car env))

(define (rest-frames env) (cdr env))

(define (no-more-frames? env) (null? env))

(define (adjoin-frame frame env) (cons frame env))

(define (set-first-frame! env new-frame)
  (set-car! env new-frame))

(define (make-frame variables values)
  (cond ((and (null? variables) (null? values)) '())
        ((null? variables)
         (error "Too many values supplied" values))
        ((null? values)
         (error "Too few values supplied" variables))
        (else
         (cons (make-binding (car variables) (car values))
               (make-frame (cdr variables) (cdr values))))))

(define (adjoin-binding binding frame)
  (cons binding frame))

(define (assq key bindings)
  (cond ((null? bindings) no-binding)
        ((eq? key (binding-variable (car bindings)))
         (car bindings))
        (else (assq key (cdr bindings)))))
```

```
(define (binding-in-frame var frame)
  (assq var frame))

(define (found-binding? b)
  (not (eq? b no-binding)))

(define no-binding nil)

(define (make-binding variable value)
  (cons variable value))

(define (binding-variable binding)
  (car binding))

(define (binding-value binding)
  (cdr binding))

(define (set-binding-value! binding value)
  (set-cdr! binding value))

(define the-empty-environment '())


; ***************************************************************************
; These functions define the syntax of Scheme-ES by providing selectors for
; the various statement and expression types, as well as tests for
; determining whether a form is a statement or an expression, and of what
; type.  Additionally, these functions are responsible for providing error
; messages in case of illegal syntax.
; ***************************************************************************

(define (stmt-type stmt)
  (if (atom? stmt)
      (error "Statement syntax error" stmt)
      (if (null? (get-prop (car stmt) 'unsugar-stmt))    ; legal keyword?
          (error "Statement syntax error" stmt)
          (car stmt))))
```

```
(define (expr-type expr)
  (if (pair? expr)
      (if (eq? (car expr) 'quote)
          atom-type
          (if (atom? (car expr))
              (if (not (null? (get-prop (car expr)
                                         'unsugar-expr)))        ; keyword?
                  (car expr)
                  application-type)
              application-type))
      (if (symbol? expr)
          (if (or (eq? expr 'nil)
                  (eq? expr 't)
                  (primitive-func? expr)
                  (primitive-proc? expr)
                  (string? expr))
              atom-type
              id-type)
          atom-type)))

(define (string? x)  ; horrible kludgey workaround due to horrible kludgey
                     ; implementation of strings in Scheme-in-Maclisp
  (if (symbol? x)
      (= (ascii (car (lisp-eval `(explode ',x)))) 34)
      nil))

(define (call-stmt-proc stmt)
  (if (atom? (cdr stmt))
      (error "CALL without procedure" stmt)
      (cadr stmt)))

(define (call-stmt-args stmt)
  (if (atom? (cdr stmt))
      (error "CALL without procedure" stmt)
      (cddr stmt)))

(define (set-stmt-loc stmt)
  (if (atom? (cdr stmt))
      (error "SET without location" stmt)
      (cadr stmt)))

(define (set-stmt-val stmt)
  (if (atom? (cddr stmt))
      (error "SET without value" stmt)
      (caddr stmt)))

(define (new-stmt-ids stmt)
  (if (atom? (cdr stmt))
      (error "NEW without identifiers" stmt)
      (cadr stmt)))
```

```
(define (new-stmt-stmts stmt)
  (if (atom? (cdr stmt))
      (error "NEW without identifiers" stmt)
      (cddr stmt)))

(define sequence-stmt-stmts cdr)

(define (atom-expr-atom expr)
  (if (quoted? expr)
      (quoted-expr-body expr)
      expr))

(define (quoted? expr)
  (if (pair? expr)
      (eq? (car expr) 'quote)
      nil))

(define (quoted-expr-body expr)
  (if (atom? (cdr expr))
      (error "QUOTE without form" expr)
      (cadr expr)))

(define (id-expr-id expr)
  expr)

(define application-expr-func car)

(define application-expr-args cdr)

(define (func-expr-ids expr)
  (if (atom? (cdr expr))
      (error "FUNC without identifiers" expr)
      (cadr expr)))

(define (func-expr-expr expr)
  (if (atom? (cddr expr))
      (error "FUNC without body" expr)
      (caddr expr)))

(define proc-expr-stmts cdr)

(define (if-expr-test expr)
  (if (atom? (cdr expr))
      (error "IF without test" expr)
      (cadr expr)))

(define (if-expr-true expr)
  (if (atom? (cddr expr))
      (error "IF without true-clause" expr)
      (caddr expr)))
```

```
(define (if-expr-false expr)
  (if (atom? (cdddr expr))
      (error "IF without false-clause" expr)
      (cadddr expr)))

(define (local-expr-stmts expr)
  (all-but-last (cdr expr)))

(define (all-but-last list)
  (if (null? (cdr list))
      '()
      (cons (car list) (all-but-last (cdr list)))))

(define (local-expr-expr expr)
  (if (atom? (cdr expr))
      (error "LOCAL without expression" expr)
      (car (last expr))))

(define local-expr-stmts-and-expr cdr)

(define (cont-expr-loc expr)
  (if (atom? (cdr expr))
      (error "CONT without location" expr)
      (cadr expr)))

(define (let-expr-pairs expr)
  (if (atom? (cdr expr))
      (error "LET without bindings" expr)
      (if (list? (cadr expr))
          (cadr expr)
          (error "LET-bindings not a list" expr))))

(define (let-expr-expr expr)
  (if (atom? (cddr expr))
      (error "LET without body" expr)
      (caddr expr)))

(define (let-stmt-stmts stmt)
  (if (atom? (cdr stmt))
      (error "LET syntax error" stmt)
      (cddr stmt)))

(define (let-pair-id pair)
  (if (atom? pair)
      (error "LET-binding syntax error" pair)
      (car pair)))

(define (let-pair-expr pair)
  (if (atom? (cdr pair))
      (error "LET-binding syntax error" pair)
      (cadr pair)))

(define letrec-expr-pairs let-expr-pairs)
(define letrec-expr-expr let-expr-expr)
```

```
(define letrec-stmt-stmts let-stmt-stmts)
(define letrec-pair-id let-pair-id)
(define letrec-pair-expr let-pair-expr)

(define (paramproc-expr-ids expr)
  (if (atom? (cdr expr))
      (error "PARAMPROC without formals" expr)
      (cadr expr)))

(define (paramproc-expr-stmts expr)
  (if (atom? (cdr expr))
      (error "PARAMPROC without formals" expr)
      (cddr expr)))

(define cond-expr-clauses cdr)

(define (cond-clause-test clause)
  (if (atom? clause)
      (error "Illegal COND clause syntax." clause)
      (car clause)))

(define (cond-clause-expr clause)
  (if (atom? clause)
      (error "Illegal COND clause syntax." clause)
      (if (atom? (cdr clause))
          (error "Illegal COND clause syntax." clause)
          (cadr clause))))

(define (cond-clause-stmts clause)
  (if (atom? clause)
      (error "Illegal COND clause syntax." clause)
      (if (atom? (cdr clause))
          (error "Illegal COND clause syntax." clause)
          (cdr clause))))

(define (stmt? form)
  (if (atom? form)
      nil
      (if (pair? (car form))
          nil
          (if (not (null? (get-prop (car form) 'exec)))
              t
              (polymorphically-stmt? form)))))

(define (polymorphically-stmt? form)
  (let ((test (get-prop (car form) 'polymorphically-stmt?)))
    (if test
        (test form)
        nil)))

(put!-prop 'if
           'polymorphically-stmt?
           (lambda (form)
             (stmt? (if-expr-true form))))
```

```
(put!-prop 'let
           'polymorphically-stmt?
           (lambda (form)
             (stmt? (let-expr-expr form))))

(put!-prop 'letrec
           'polymorphically-stmt?
           (lambda (form)
             (stmt? (letrec-expr-expr form))))

(put!-prop 'cond
           'polymorphically-stmt?
           (lambda (form)
             (let ((clauses (cond-expr-clauses form)))
               (if (null? clauses)
                   nil
                   (stmt? (cond-clause-expr (car clauses)))))))


; ***************************************************************************
; This section of the code is a long, boring definition of all the
; Scheme-ES primitives.  They are all identical to their Scheme cousins
; except MUTABLE-CONS, EQ?, ERROR, and MAPCAN.  The handling of MUTABLE-
; CONS, EQ?, and ERROR is described in the text of the paper; MAPCAN uses
; APPEND instead of CONC!.  The only additional primitives are CALLABLE?,
; which is to procedures as APPLICABLE? is to functions, SET-PRINT-DEPTH
; and SET-PRINT-BREADTH, which replace the corresponding globals, and
; STRING?, which does the obvious thing.  (In Scheme, strings are symbols,
; which caused this interpreter writer some pain.)
; ***************************************************************************

(define effects-allowed? t)  ; Not inside a LOCAL?

(define (primitive-proc? x)
  (if (symbol? x)
      (not (null? (get-prop x 'primitive-proc)))
      nil))

(define (primitive-func? x)
  (if (symbol? x)
      (not (null? (get-prop x 'primitive-func)))
      nil))

(define (call-primitive proc store)
  ((get-prop proc 'primitive-proc) store))

(define (apply-primitive func args env store)
  ((get-prop func 'primitive-func) args env store))
```

```
(define (import-func Scheme-ES-name func)
  (put!-prop Scheme-ES-name
       'primitive-func
       (lambda (args env store)
         (let ((args (mapcar (lambda (arg)
                               (kernel-eval arg env store))
                             args)))
           (let ((err-objs (filter error-object? args)))
             (if (null? err-objs)
                 (apply func args)
                 (car err-objs)))))))   ; arbitrarily pick first one

(define (filter predicate list)
  (cond ((null? list)
         '())
        ((predicate (car list))
         (cons (car list) (filter predicate (cdr list))))
        (else
         (filter predicate (cdr list)))))

(define (import-proc Scheme-ES-name proc)
  (put!-prop Scheme-ES-name
             'primitive-proc
             (lambda (store)
               (proc))))

(define (scheme-ES-and . args)  ; in maclisp-scheme AND is a special form
  (if (null? args)
      t
      (and (car args) (apply scheme-ES-and (cdr args)))))

(define (scheme-ES-or . args)   ; in maclisp-scheme OR is a special form
  (if (null? args)
      nil
      (or (car args) (apply scheme-ES-or (cdr args)))))

(import-func 'and scheme-ES-and)
(import-func 'or scheme-ES-or)
(import-func 'not not)
(import-func 'nil? nil?)
(import-func '+ +)
(import-func '1+ 1+)
(import-func '- -)
(import-func '-1+ -1+)
(import-func '* *)
(import-func '/ /)
(import-func 'quotient quotient)
(import-func 'mod mod)
(import-func 'remainder remainder)
(import-func 'integer-divide integer-divide)
(import-func 'gcd gcd)
(import-func 'abs abs)
(import-func 'floor floor)
(import-func 'ceiling ceiling)
```

```
(import-func 'truncate truncate)
(import-func 'round round)
(import-func 'max max)
(import-func 'min min)
(import-func 'sin sin)
(import-func 'asin asin)
(import-func 'cos cos)
(import-func 'acos acos)
(import-func 'tan tan)
(import-func 'atan atan)
(import-func 'log log)
(import-func 'exp exp)
(import-func 'expt expt)
(import-func 'sqrt sqrt)
(import-func 'number? number?)
(import-func 'integer? integer?)
(import-func 'odd? odd?)
(import-func 'even? even?)
(import-func 'zero? zero?)
(import-func 'negative? negative?)
(import-func 'positive? positive?)
(import-func '= =)
(import-func '> >)
(import-func '>= >=)
(import-func '< <)
(import-func '<= <=)
(import-func 'null? null?)
(import-func 'alphaless? alphaless?)
(import-proc 'newline (lambda ()
                        (if effects-allowed?
                            (newline)
                            (error "Can't do I/O inside a LOCAL"
                                   'newline))))
(import-func 'char char)
(import-func 'ascii ascii)
(import-func 'peekch peekch)
(import-func 'tyipeek tyipeek)
(import-func '%in (lambda () last-read-in))
(import-func '%out (lambda () last-printed-out))
(import-func 'string? string?)
```

```
(define (Scheme-ES-eq? x y)
  (cond ((or (compound-func? x)
             (compound-func? y)
             (compound-proc? x)
             (compound-proc? y))
         (error "Can't test equality of compound procedures and functions"
                (list (unparse-s-expr x) (unparse-s-expr y))))
        ((eq? x y)
         t)
        ((and (number? x) (number? y))
         (= x y))
        ((and (immutable-cons? x) (immutable-cons? y))
         (and (Scheme-ES-eq? (cons-cell-car x) (cons-cell-car y))
              (Scheme-ES-eq? (cons-cell-cdr x) (cons-cell-cdr y))))
        (else
         nil)))

(import-func 'eq? Scheme-ES-eq?)

(import-func 'random (lambda (first . rest)
                       (let ((proc (gensym 'random))
                             (result (if (null? rest)
                                         first
                                         (car rest)))
                             (args (if (null? rest)
                                       '()
                                       (list first))))
                         (put!-prop proc
                                    'primitive-proc
                                    (lambda (store)
                                      (randomize randomize-to)
                                      (assign! result
                                               (let ((r (apply random
                                                               args)))
                                                 (set! randomize-to r)
                                                 r)
                                               store)))
                         proc)))

(import-func 'randomize (lambda (q)
                          (let ((proc (gensym 'randomize)))
                            (import-proc proc (lambda ()
                                                (set! randomize-to q)))
                            proc)))

(define randomize-to nil)  ; exists for sake of LOCAL

(import-func 'extend? (lambda (obj)
                        (or (primitive-func? obj)
                            (compound-func? obj)
                            (primitive-proc? obj)
                            (compound-proc? obj))))
```

```
(import-func 'applicable? (lambda (obj)
                            (or (primitive-func? obj)
                                (compound-func? obj))))

(import-func 'callable? (lambda (obj)
                          (or (primitive-proc? obj)
                              (compound-proc? obj))))

(import-func 'read (lambda (result)
                     (let ((proc (gensym 'read)))
                       (put!-prop proc
                                  'primitive-proc
                                  (lambda (store)
                                    (if (not effects-allowed?)
                                        (error
                                          "Can't do I/O inside a LOCAL"
                                          'read)
                                        (assign result
                                                (let ((val (parse-s-expr
                                                             (read))))
                                                  (set! last-read-in val)
                                                  val)
                                                store))))
                       proc)))

(define (parse-s-expr s-expr)
  (if (pair? s-expr)
      (immutable-cons (parse-s-expr (car s-expr))
                      (parse-s-expr (cdr s-expr)))
      s-expr))

(define last-read-in nil)

(import-func 'print (lambda (val)
                      (let ((proc (gensym 'print)))
                        (import-proc proc
                                     (lambda ()
                                       (if (not effects-allowed?)
                                           (error
                                             "Can't do I/O inside a LOCAL"
                                             'print)
                                           (sequence
                                             (set! last-printed-out val)
                                             (print (unparse-s-expr
                                                      val))))))
                        proc)))
```

```
(define (unparse-s-expr x)
  (cond ((cons-cell? x)
         (cons (unparse-s-expr (cons-cell-car x))
               (unparse-s-expr (cons-cell-cdr x))))
        ((compound-func? x)
         `(<COMPOUND-FUNCTION> ,(func-formals x) ,(func-body x)))
        ((compound-proc? x)
         `(<COMPOUND-PROCEDURE> . ,(proc-body x)))
        (else
         x)))

(define last-printed-out nil)

(import-func 'princ (lambda (val)
                      (let ((proc (gensym 'princ)))
                        (import-proc proc
                                     (lambda ()
                                       (if (not effects-allowed?)
                                           (error
                                             "Can't do I/O inside a LOCAL"
                                             'princ)
                                           (sequence
                                             (set! last-printed-out val)
                                             (princ (unparse-s-expr
                                                      val))))))
                        proc)))

(import-func 'tyi (lambda (result)
                    (let ((proc (gensym 'tyi)))
                      (put!-prop proc
                                 'primitive-proc
                                 (lambda (store)
                                   (if (not effects-allowed?)
                                       (error
                                         "Can't do I/O inside a LOCAL"
                                         'tyi)
                                       (assign! result
                                                (tyi)
                                                store))))
                      proc)))

(import-func 'tyo (lambda (n)
                    (let ((proc (gensym 'tyo)))
                      (import-proc proc
                                   (lambda ()
                                     (if (not effects-allowed?)
                                         (error
                                           "Can't do I/O inside a LOCAL"
                                           'tyo)
                                         (tyo n))))
                      proc)))
```

```
(import-func 'readch (lambda (result)
                       (let ((proc (gensym 'readch)))
                         (put!-prop proc
                                    'primitive-proc
                                    (lambda (store)
                                      (if (not effects-allowed?)
                                          (error
                                            "Can't do I/O inside a LOCAL"
                                            'readch)
                                          (assign! result
                                                   (readch)
                                                   store))))
                         proc)))

(import-func 'set-print-breadth (lambda (n)
                                  (let ((proc (gensym
                                                'set-print-breadth)))
                                    (import-proc proc
                                                 (lambda ()
                                                   (set! *print-breadth*
                                                         n)))
                                    proc)))

(import-func 'set-print-depth (lambda (n)
                                (let ((proc (gensym 'set-print-depth)))
                                  (import-proc proc
                                               (lambda ()
                                                 (set! *print-depth* n)))
                                  proc)))

(define (immutable-cons x y)
  `(immutable-cons ,x ,y))

(define (immutable-cons? x)
  (if (pair? x)
      (eq? (car x) 'immutable-cons)
      nil))

(define cons-cell-car cadr)

(define cons-cell-cdr caddr)

(define (cons-cell? x)
  (or (immutable-cons? x)
      (mutable-cons? x)))

(import-func 'cons immutable-cons)

(define (Scheme-ES-cons* x y . rest)
  (if (null? rest)
      (immutable-cons x y)
      (immutable-cons x (apply Scheme-ES-cons* (cons y rest)))))

(import-func 'cons* Scheme-ES-cons*)
```

```
(define (Scheme-ES-list . args)
  (if (null? args)
      '()
      (immutable-cons (car args) (apply Scheme-ES-cons (cdr args)))))

(import-func 'list Scheme-ES-list)

(import-func 'car cons-cell-car)
(import-func 'cdr cons-cell-cdr)
(import-func 'caar (lambda (x) (caar (unparse-s-expr x))))
(import-func 'cadr (lambda (x) (cadr (unparse-s-expr x))))
(import-func 'cdar (lambda (x) (cdar (unparse-s-expr x))))
(import-func 'cddr (lambda (x) (cddr (unparse-s-expr x))))
(import-func 'caaar (lambda (x) (caaar (unparse-s-expr x))))
(import-func 'caadr (lambda (x) (caadr (unparse-s-expr x))))
(import-func 'cadar (lambda (x) (cadar (unparse-s-expr x))))
(import-func 'caddr (lambda (x) (caddr (unparse-s-expr x))))
(import-func 'cdaar (lambda (x) (cdaar (unparse-s-expr x))))
(import-func 'cdadr (lambda (x) (cdadr (unparse-s-expr x))))
(import-func 'cddar (lambda (x) (cddar (unparse-s-expr x))))
(import-func 'cdddr (lambda (x) (cdddr (unparse-s-expr x))))
(import-func 'caaaar (lambda (x) (caaaar (unparse-s-expr x))))
(import-func 'caaadr (lambda (x) (caaadr (unparse-s-expr x))))
(import-func 'caadar (lambda (x) (caadar (unparse-s-expr x))))
(import-func 'caaddr (lambda (x) (caaddr (unparse-s-expr x))))
(import-func 'cadaar (lambda (x) (cadaar (unparse-s-expr x))))
(import-func 'cadadr (lambda (x) (cadadr (unparse-s-expr x))))
(import-func 'caddar (lambda (x) (caddar (unparse-s-expr x))))
(import-func 'cadddr (lambda (x) (cadddr (unparse-s-expr x))))
(import-func 'cdaaar (lambda (x) (cdaaar (unparse-s-expr x))))
(import-func 'cdaadr (lambda (x) (cdaadr (unparse-s-expr x))))
(import-func 'cdadar (lambda (x) (cdadar (unparse-s-expr x))))
(import-func 'cdaddr (lambda (x) (cdaddr (unparse-s-expr x))))
(import-func 'cddaar (lambda (x) (cddaar (unparse-s-expr x))))
(import-func 'cddadr (lambda (x) (cddadr (unparse-s-expr x))))
(import-func 'cdddar (lambda (x) (cdddar (unparse-s-expr x))))
(import-func 'cddddr (lambda (x) (cddddr (unparse-s-expr x))))

(define (mutable-cons x y)
  `(mutable-cons ,x ,y))

(define (mutable-cons? x)
  (if (pair? x)
      (eq? (car x) 'mutable-cons)
      nil))

(define (Scheme-ES-set-car! pair newcar)
  (if (mutable-cons? pair)
      (set!-car (cdr pair) newcar)
      (error "can't set-car!" pair)))
```

```
(define (Scheme-ES-set-cdr! pair newcdr)
  (if (mutable-cons? pair)
      (set!-car (cddr pair) newcdr)
      (error "can't set-cdr!" pair)))

(import-func 'mutable-cons (lambda (x y result)
                             (let ((proc (gensym 'mutable-cons)))
                               (put!-prop proc
                                          'primitive-proc
                                          (lambda (store)
                                            (assign! result
                                                     (mutable-cons x y)
                                                     store)))
                               proc)))

(import-func 'set-car! (lambda (pair newcar)
                         (let ((proc (gensym 'set-car!)))
                           (import-proc proc
                                        (lambda ()
                                          (Scheme-ES-set-car! pair
                                                              newcar)))
                           proc)))

(import-func 'set-cdr! (lambda (pair newcdr)
                         (let ((proc (gensym 'set-cdr!)))
                           (import-proc proc
                                        (lambda ()
                                          (Scheme-ES-set-cdr! pair
                                                              newcdr)))
                           proc)))

(import-func 'atom? (lambda (x) (not (cons-cell? x))))

(import-func 'symbol? (lambda (x)
                        (and (symbol? x)
                             (not (string? x))
                             (not (primitive-proc? x))
                             (not (primitive-func? x)))))

(import-func 'pair? cons-cell?)

(import-func 'list? (lambda (x) (or (null? x) (cons-cell? x))))

(define (Scheme-ES-equal? x y)
  (cond ((Scheme-ES-eq? x y)
         t)
        ((and (cons-cell? x) (cons-cell? y))
         (and (Scheme-ES-eq? (cons-cell-car x) (cons-cell-car y))
              (Scheme-ES-equal? (cons-cell-cdr x) (cons-cell-cdr y))))
        (else
         nil)))

(import-func 'equal? Scheme-ES-equal?)
```

```
(define (Scheme-ES-nthcdr n l)
  (if (= n 0)
      l
      (Scheme-ES-nthcdr (-1+ n) (cons-cell-cdr l))))

(import-func 'nthcdr Scheme-ES-nthcdr)

(import-func 'nth (lambda (n l) (cons-cell-car (Scheme-ES-nthcdr n l))))

(define (Scheme-ES-last l)
  (if (null? (cons-cell-cdr l))
      l
      (Scheme-ES-last (cons-cell-cdr l))))

(import-func 'last Scheme-ES-last)

(define (Scheme-ES-length l)
  (if (null? l)
      0
      (1+ (Scheme-ES-length (cons-cell-cdr l)))))

(import-func 'length Scheme-ES-length)

(define (Scheme-ES-append l1 l2)
  (if (null? l1)
      l2
      (immutable-cons (cons-cell-car l1)
                      (Scheme-ES-append (cons-cell-cdr l1) l2))))

(import-func 'append Scheme-ES-append)

(define (Scheme-ES-reverse l)
  (define (reverse-and-append l1 l2)
    (if (null? l1)
        l2
        (reverse-and-append (cons-cell-cdr l1)
                            (immutable-cons (cons-cell-car l1) l2))))
  (reverse-and-append l '()))

(import-func 'reverse Scheme-ES-reverse)

(import-func 'conc!
             (lambda (l1 l2)
               (let ((proc (gensym 'conc!)))
                 (import-proc proc
                              (lambda ()
                                (Scheme-ES-set-cdr! (Scheme-ES-last l1)
                                                    l2)))
                 proc)))
```

```
(define (Scheme-ES-mapcar f l env store)
  (cond ((error-object? f)
         f)
        ((error-object? l)
         l)
        ((null? l)
         '())
        (else
         (immutable-cons (kernel-eval '(f (car l))
                                      (extend-environment '(f l)
                                                          `(,f ,l)
                                                          '())
                                      store)
                         (Scheme-ES-mapcar f
                                           (cons-cell-cdr l)
                                           env
                                           store)))))

(put!-prop 'mapcar
           'primitive-func
           (lambda (args env store)
             (Scheme-ES-mapcar (kernel-eval (car args) env store)
                               (kernel-eval (cadr args) env store)
                               env
                               store)))

(define (Scheme-ES-mapcan f l env store)
  (cond ((error-object? f)
         f)
        ((error-object? l)
         l)
        ((null? l)
         '())
        (else
         (Scheme-ES-append (kernel-eval '(f (car l))
                                        (extend-environment '(f l)
                                                            `(,f ,l)
                                                            '())
                                        store)
                           (Scheme-ES-mapcan f
                                             (cons-cell-cdr l)
                                             env
                                             store)))))

(put!-prop 'mapcan
           'primitive-func
           (lambda (args env store)
             (Scheme-ES-mapcan (kernel-eval (car args) env store)
                               (kernel-eval (cadr args) env store)
                               env
                               store)))
```

```
(define (Scheme-ES-memq obj list)
  (cond ((null? list)
         nil)
        ((Scheme-ES-eq? obj (cons-cell-car list))
         list)
        (else
         (Scheme-ES-memq obj (cons-cell-cdr list)))))

(import-func 'memq Scheme-ES-memq)

(define (Scheme-ES-member obj list)
  (cond ((null? list)
         nil)
        ((Scheme-ES-equal? obj (cons-cell-car list))
         list)
        (else
         (Scheme-ES-member obj (cons-cell-cdr list)))))

(import-func 'member Scheme-ES-member)

(define (Scheme-ES-assq obj alist)
  (cond ((null? alist)
         nil)
        ((Scheme-ES-eq? obj (cons-cell-car (cons-cell-car alist)))
         (cons-cell-car alist))
        (else
         (Scheme-ES-assq obj (cons-cell-cdr alist)))))

(import-func 'assq Scheme-ES-assq)

(define (Scheme-ES-assoc obj alist)
  (cond ((null? alist)
         nil)
        ((Scheme-ES-equal? obj (cons-cell-car (cons-cell-car alist)))
         (cons-cell-car alist))
        (else
         (Scheme-ES-assoc obj (cons-cell-cdr alist)))))

(import-func 'assoc Scheme-ES-assoc)

(define (make-error-object message irritant)
  `(error-object ,message ,irritant))

(define (error-object? x)
  (if (pair? x)
      (eq? (car x) 'error-object)
      nil))

(define error-object-message cadr)

(define error-object-irritant caddr)

(import-func 'error make-error-object)
```

Appendix B: Composing Mutable-Object Constructors: Constraint-Propagation

The section on composing mutable-data-structure constructors uses mutable cons-cells for its example, but it was inspired by a more complicated example from Abelson and Sussman. After presenting an object-oriented constraint-propagation system[21], they pose Exercise 3.37[22], which is a typical example of the composition problem explained in the text (with message-passing objects for the mutable data-structures). The goal is to be able to avoid the step by step computation and explicit allocation of temporaries of CENTIGRADE-FAHRENHEIT-CONVERTER below, and instead write the converter functionally, as in C-F-CONVERTER below. In addition to these two procedures, the solution by abstraction is shown below, and also a example usage with its output.

```
(define (centigrade-fahrenheit-converter c f)[23]
  (new (u v w x y)
    (call make-connector u)
    (call make-connector v)
    (call make-connector w)
    (call make-connector x)
    (call make-connector y)
    (call multiplier c (cont w) (cont u))
    (call multiplier (cont v) (cont x) (cont u))
    (call adder (cont v) (cont y) f)
    (call constant 9 (cont w))
    (call constant 5 (cont x))
    (call constant 32 (cont y))))
```

```
(define (c-f-converter c)[24]
  (c+ (c* (c/ (cvalue 9) (cvalue 5))
          c)
      (cvalue 32)))
```

[21] Abelson and Sussman, pp. 230-242.
[22] Abelson and Sussman, p. 241.
[23] Abelson and Sussman, pp. 232-233.
[24] Abelson and Sussman, p. 241.

```
(define (c+ xconstructor yconstructor)
  (paramproc (sum)
    (new (x y)
      (call xconstructor x)
      (call yconstructor y)
      (call make-connector sum)
      (call adder (cont x) (cont y) (cont sum)))))

(define (c* xconstructor yconstructor)
  (paramproc (product)
    (new (x y)
      (call xconstructor x)
      (call yconstructor y)
      (call make-connector product)
      (call multiplier (cont x) (cont y) (cont product)))))

(define (c- xconstructor yconstructor)
  (paramproc (difference)
    (new (x y)
      (call xconstructor x)
      (call yconstructor y)
      (call make-connector difference)
      (call adder (cont difference) (cont y) (cont x)))))

(define (c/ xconstructor yconstructor)
  (paramproc (quot)
    (new (x y)
      (call xconstructor x)
      (call yconstructor y)
      (call make-connector quot)
      (call multiplier (cont quot) (cont y) (cont x)))))

(define (cvalue value)
  (paramproc (result)
    (call make-connector result)
    (call constant value (cont result))))

(define (cconnector connector)
  (paramproc (result)
    (set result connector)))

(define (cset loc constructor)
  (call constructor loc))
```

```
(new (C F)
  (call make-connector C)
  (call cset F (c-f-converter (cconnector (cont C))))
  (call probe "centigrade temp" (cont C))
  (call probe "Fahrenheit temp" (cont F))
  (call set-value! (cont C) 25 'user)
  (call forget-value! (cont C) 'user)
  (call set-value! (cont F) 212 'user))

Probe: centigrade temp = 25
Probe: Fahrenheit temp = 77
Probe: centigrade temp = ?
Probe: Fahrenheit temp = ?
Probe: Fahrenheit temp = 212
Probe: centigrade temp = 100
```