```
(set! *fun*          'anything)
(set! *arg1*         'anything)
(set! *arg2*         'anything)
(stack-push finish-pc)                    ;pc for FINISH
(set! *pc*           pc)
(call/cc (lambda (exit)
           (set! *exit* exit)
           (run) )) )
```

7.5.7 Catching Our Breath

To finish off this long section about instructions, we'll look at the final form of the program we saw at the beginning of the chapter, as byte code, once it has been disassembled to be more readable. See Figure 7.5.

7.6 Continuations

We've already shown that `call/cc` is a kind of magic operator that reifies the evaluation context, turning the context into an object that can be invoked. Now we'll demystify its implementation. The following implementation is canonical. You'll find more efficient but more complicated ones in [CHO88, HDB90, MB93].

The evaluation context is made up of the stack and nothing but the stack. In effect, the registers `*fun*`, `*arg1*`, and `*arg2*` play only temporary roles; in no case can they be captured. (You can't call[10] `call/cc` nor anything else while they are active.)

The register `*val*` transmits values submitted to continuations and thus is not anything to save. The register `*env*` need not be saved either because `call/cc` is not a function that we integrate there; in fact, the environment has already been saved because of the call to `call/cc`. Consequently, there is only the stack to save, and to do so, we'll use these functions:

```
(define (save-stack)
  (let ((copy (make-vector *stack-index*)))
    (vector-copy! *stack* copy 0 *stack-index*)
    copy ) )
(define (restore-stack copy)
  (set! *stack-index* (vector-length copy))
  (vector-copy! copy *stack* 0 *stack-index*) )
(define (vector-copy! old new start end)
  (let copy ((i start))
    (when (< i end)
          (vector-set! new i (vector-ref old i))
          (copy (+ i 1)) ) ) )
```

Continuations will have their own class and their own special calling protocol. When a continuation is invoked, it restores the stack, puts the value received into the register `*val*`, and then branches (by a **RETURN**) to the address contained on

10. That statement may be false if we have to respond to asynchronous calls, such as UN*X signals. In that case, it's better to set a flag that we test regularly from time to time as in [Dev85].

```
(CREATE-CLOSURE 2)
(SHORT-GOTO 59)
(ARITY=?4)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(CONSTANT0)
(POP-ARG1)
(CALL2-=)
(SHORT-JUMP-FALSE 10)
(SHALLOW-ARGUMENT-REF2)
(PUSH-VALUE)
(CONSTANT1)
(PUSH-VALUE)
(ALLOCATE-FRAME2)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(SHORT-GOTO 39)
(SHALLOW-ARGUMENT-REF1)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(CONSTANT1)
(POP-ARG1)
(CALL2--)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF1)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(SHORT-GOTO 19)
(ARITY=?2)
(EXTEND-ENV)
(DEEP-ARGUMENT-REF 1 2)
(PUSH-VALUE)
(DEEP-ARGUMENT-REF 1 0)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF0)
(POP-ARG1)
(CALL2-*)
(PUSH-VALUE)
(ALLOCATE-FRAME2)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME4)
(POP-FRAME!2)
(POP-FRAME!1)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME2)
(POP-FRAME!0)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(SHORT-NUMBER 5)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(SHORT-GOTO 4)
(ARITY=?2)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF0)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME4)
(POP-FRAME!2)
(POP-FRAME!1)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
```

Figure 7.5 Compilation result

top of the stack. Since `call/cc` can be called only by `(FUNCTION-INVOKE)`, the following instruction will necessarily be a `(RESTORE-ENV)`. (You can verify that in the definition of `REGULAR-CALL`). [see p. 229]

```
(define-class continuation Object
  ( stack ) )
(define-method (invoke (f continuation) tail?)
  (if (= (+ 1 1) (activation-frame-argument-length *val*))
      (begin
        (restore-stack (continuation-stack f))
        (set! *val* (activation-frame-argument *val* 0))
        (set! *pc* (stack-pop)) )
      (signal-exception #f (list "Incorrect arity" 'continuation)) ) )
```

For the invocation of a continuation to succeed, the stack must have a special structure built by `call/cc`. Building that structure is a subtle task because the capture of the continuation must not consume the stack. That is, in the form `(call/cc f)`, the function f must be called in tail[11] position. The following definition accomplishes that. It allocates an activation record, fills in the reified continuation, and calls its argument `f` with it.

```
(definitial call/cc
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let ((f (activation-frame-argument *val* 0))
                 (frame (allocate-activation-frame (+ 1 1))) )
             (set-activation-frame-argument!
              frame 0 (make-continuation (save-stack)) )
             (set! *val* frame)
             (set! *fun* f)              ;useful for debug
             (invoke f #t) )
           (signal-exception #t (list "Incorrect arity"
                                      'call/cc )) ) ) ) ) )
```

In conclusion, the canonical implementation of `call/cc` costs one copy of the stack for each reification and another copy at every invocation of a continuation. Better strategies exist, but they are more complicated to implement. The main idea of most of these strategies is that when a continuation is reified, it stands a good chance of happening again, either in whole or in part, according to [Dan87], so strategies that allow captures to be shared should be favored.

7.7 Escapes

Even if the cost of `call/cc` can be reduced, it wouldn't be fair not to show how to implement simple escapes. We've decided to implement the special form `bind-exit`. [see p. 101] It's present in Dylan and analogous to `let/cc` in EuLisp, to `block/return-from` in Common Lisp, and to `escape` in Vlisp [Cha80].

11. That's not necessary in Scheme.

In contrast to the spirit of Scheme, we've opted for a new special form rather than a function for two main reasons: first, introducing a new special form is slightly more simple because in doing so, we don't have to conform to the structure of the stack as dictated by the function calling protocol; second, we can thus show the entire compiler in cross-section. If you need yet another reason, remember that a function and a special form are equally powerful here since we can write one in terms of the other.

The form **bind-exit** has the following syntax:

`(bind-exit (`*variable*`)` *forms* `...)` Dylan

The *variable* is bound to the continuation of the form **bind-exit**; then the body of the form is evaluated. The captured continuation can be used only during that evaluation. If **bind-exit** is available to us, we can define the function **call/ep** (for *call-with-exit-procedure*) and conversely.

```
(define (call/ep f)
  (bind-exit (k) (f k)) )
```

`(bind-exit (`*k*`)` *body*`)` ≡ `(call/ep (lambda (`*k*`)` *body*`))`

Escapes are represented by objects of the class **escape**. They have a unique field to designate the height of the stack where the evaluation of the form **bind-exit** begins.

```
(define-class escape Object
  ( stack-index ) )
```

To define a new special form, we must first add a clause to the function **meaning**, the lexical analyzer of forms to compile. That additional clause will recognize these new forms, so we'll add the following clause to **meaning**:

```
... ((bind-exit) (meaning-bind-exit (caadr e) (cddr e) r tail?)) ...
```

Then we'll define the pretreatment function for those forms, like this:

```
(define (meaning-bind-exit n e+ r tail?)
  (let* ((r2 (r-extend* r (list n)))
         (m+ (meaning-sequence e+ r2 #t)) )
    (ESCAPER m+) ) )
```

The pretreatment consists of making a sequence from the body of the form **bind-exit**; that sequence will be pretreated in a lexical environment extended by the local variable that introduces **bind-exit**. The function **ESCAPER** (all in upper case letters) takes care of generating the code. For that function, we invent two new instructions: **PUSH-ESCAPER** and **POP-ESCAPER**.

```
(define (ESCAPER m+)
  (append (PUSH-ESCAPER (+ 1 (length m+))) m+ (RETURN) (POP-ESCAPER)) )
(define (POP-ESCAPER) (list 250))
(define escape-tag (list '*ESCAPE*))
(define (PUSH-ESCAPER offset) (list 251 offset))

(define-instruction (POP-ESCAPER) 250
  (let* ((tag (stack-pop))
         (escape (stack-pop)) )
    (restore-environment) ) )
```

```
(define-instruction (PUSH-ESCAPER offset) 251
  (preserve-environment)
  (let* ((escape (make-escape (+ *stack-index* 3)))
         (frame (allocate-activation-frame 1)) )
    (set-activation-frame-argument! frame 0 escape)
    (set! *env* (sr-extend* *env* frame))
    (stack-push escape)
    (stack-push escape-tag)
    (stack-push (+ *pc* offset)) ) )
```

You can see how the instruction **PUSH-ESCAPER** works in Figure 7.6. Here's what it does:

1. it saves the current environment on the stack;
2. it allocates a valid escape indicating the third word above the current top of the stack;
3. it allocates an activation record to enrich the current lexical environment; (its unique field contains the escape);
4. pushes this escape on the stack finally and then puts on that strange flag **(*ESCAPE*)**, followed by an address indicating the **(POP-ESCAPER)** instruction that follows.

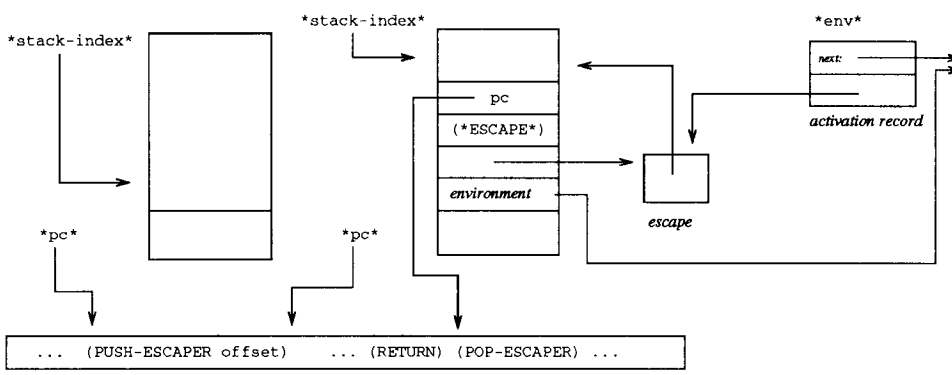


Figure 7.6 The stack before and after **PUSH-ESCAPER**

The method for invoking escapes is complicated by the fact that we have to check whether or not the escape is valid. An escape is valid if the height of the stack is greater than the height saved in the escape; if there really is an escape at that place in the stack; and if that escape really is the one we're interested in. All those conditions are verified in constant time by the function **escape-valid?**, like this:

```
(define-method (invoke (f escape) tail?)
  (if (= (+ 1 1) (activation-frame-argument-length *val*))
      (if (escape-valid? f)
          (begin (set! *stack-index* (escape-stack-index f))
                 (set! *val* (activation-frame-argument *val* 0))
```

```
                (set! *pc* (stack-pop)) )
            (signal-exception #f (list "Escape out of extent" f)) )
        (signal-exception #f (list "Incorrect arity" 'escape)) ) )
(define (escape-valid? f)
  (let ((index (escape-stack-index f)))
    (and (>= *stack-index* index)
         (eq? f (vector-ref *stack* (- index 3)))
         (eq? escape-tag (vector-ref *stack* (- index 2))) ) ) )
```

When an escape is invoked, it first verifies whether it is valid; then it takes the level of the stack back to the level prevailing at the beginning of the evaluation of the body of the form `bind-exit`; it puts the value provided into the register `*val*`; then it carries out the equivalent of a `(RETURN)` resetting the program counter. The following instruction will thus be `(POP-ESCAPER)`, which pops the stack, restores the saved environment, and exits from the form `bind-exit`.

If no escape is called during the body of the form `bind-exit`, the same scenario applies: the `(RETURN)` that precedes `(POP-ESCAPER)` removes the address from the stack makes it possible to branch to the same instruction in the same configuration of the stack as during the call to the escape.

Getting into the form `bind-exit` costs two object allocations and four places on the stack. Invoking an escape costs one validity test and a few register moves. We explicitly allocated the escape because it may happen that the variable bound by the form `bind-exit` might be captured by a closure, and being captured by a closure confers an indefinite extent on the bound variable. In the opposite case, we can avoid allocating the escape and simply keep the pointer to the stack in a register. The implementation that we just explained is quite conservative, we'll admit, and not very efficient as compared with the results of a good compilation. Nevertheless, it's more efficient than the canonical implementation of `call/cc`.

The special form `bind-exit` (along with the dynamic variables of the next section) makes it easier to program analogues of `catch/throw`. [see p. 77] In contrast, if the language provides an `unwind-protect` form, then we would have to look again at the speed of `bind-exit` because every escape would have to evaluate the cleaners associated with embedding `unwind-protect`. Those cleaners would be determined by inspecting the stack between the points of departure and arrival.

7.8 Dynamic Variables

We've already often written that dynamic variables correspond to a significant idea. We implemented them using deep binding. Like before [see p. 167], we'll assume that we have two new special forms to create and refer to dynamic variables. Here's the syntax of those forms:

`(dynamic-let (`*variable value*`) `*body...*`)`
`(dynamic `*variable*`)`

The form `dynamic-let` binds a variable and value during the evaluation of its body. We can get the value of the variable by means of the form `dynamic`. Of course, it would be an error to ask for the value of a variable that has not been bound.

For those reasons, we'll add two new clauses to **meaning**, our syntactic analyzer:

```
... ((dynamic)      (meaning-dynamic-reference (cadr e) r tail?))
    ((dynamic-let) (meaning-dynamic-let (car (cadr e))
                                        (cadr (cadr e))
                                        (cddr e) r tail? )) ...
```

We'll also associate the necessary pretreatments with them, like this:

```
(define (meaning-dynamic-let n e e+ r tail?)
  (let ((index (get-dynamic-variable-index n))
        (m (meaning e r #f))
        (m+ (meaning-sequence e+ r #f)) )
    (append m (DYNAMIC-PUSH index) m+ (DYNAMIC-POP)) ) )
(define (meaning-dynamic-reference n r tail?)
  (let ((index (get-dynamic-variable-index n)))
    (DYNAMIC-REF index) ) )
```

Those pretreatments use these three new generators:

```
(define (DYNAMIC-PUSH index) (list 242 index))
(define (DYNAMIC-POP)        (list 241))
(define (DYNAMIC-REF index)  (list 240 index))
```

Those three generators correspond to three new instructions in our virtual machine, namely:

```
(define-instruction (DYNAMIC-PUSH index) 242
  (push-dynamic-binding index *val*) )
(define-instruction (DYNAMIC-POP) 241
  (pop-dynamic-binding) )
(define-instruction (DYNAMIC-REF index) 240
  (set! *val* (find-dynamic-value index)) )
```

Now how do we represent the environment for dynamic variables? The first idea that comes to mind is to invent a new register, say, ***dynenv***, that permanently points to an association list pairing dynamic variables with their values. Well, actually, this "list" is not a real list, but rather a few *frames*, that is, regions chained together in the stack. Unfortunately, with this idea we've augmented the machine state since the contents of the register ***dynenv*** now have to be saved, too, by **PRESERVE-ENV**, and of course they have to be restored by **RESTORE-ENV**. Since those two instructions occur quite frequently already, adding the register ***dynenv*** will be costly even if we never use it. An important rule for us is that only users should pay for services; as a corollary, those who never use a service shouldn't have to pay for it. From that point of view, adding another register looks like a bad idea.

Rather than maintain a register, we give the means of establishing information about dynamic variables only to those who use them and only if they need them. The cost will thus be greater, but at least this way of doing things will not penalize those who don't need it. The environment will be implemented as frames in the stack, but each of those frames will be preceded by a special label identifying it, as in Figure 7.7. The function **search-dynenv-index** provides information that we would have been able to find in that register we decided not to add.

```
(define dynenv-tag (list '*dynenv*))
```

```
(define (search-dynenv-index)
  (let search ((i (- *stack-index* 1)))
    (if (< i 0) i
        (if (eq? (vector-ref *stack* i) dynenv-tag)
            (- i 1)
            (search (- i 1)) ) ) ) )
(define (pop-dynamic-binding)
  (stack-pop)
  (stack-pop)
  (stack-pop)
  (stack-pop) )
(define (push-dynamic-binding index value)
  (stack-push (search-dynenv-index))
  (stack-push value)
  (stack-push index)
  (stack-push dynenv-tag) )
```

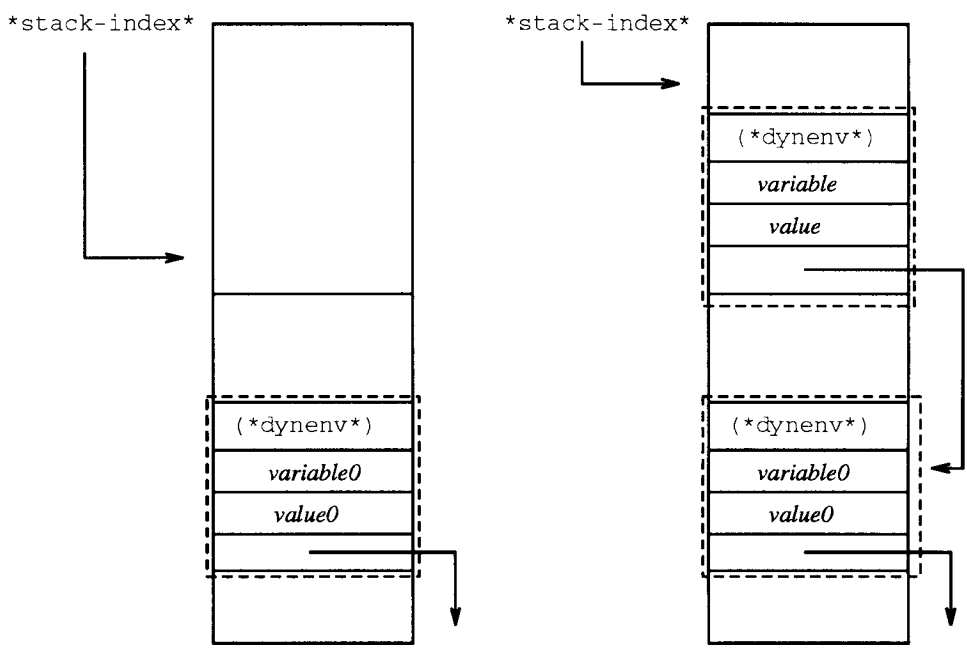


Figure 7.7 Stack before and after `DYNAMIC-PUSH`

The one obscure point that we still have to clear up is this: how do we refer to dynamic variables? A reference like `(dynamic foo)` must be compiled into bytes, so that rules out referring directly to the symbol `foo`. We've already numbered mutable global variables as they appeared, so in the same way, we'll number dynamic variables by means of `get-dynamic-variable-index`, like this:

```
(define *dynamic-variables* '())
(define (get-dynamic-variable-index n)
```

```
(let ((where (memq n *dynamic-variables*)))
  (if where (length where)
      (begin
        (set! *dynamic-variables* (cons n *dynamic-variables*))
        (length *dynamic-variables*) ) ) ) )
```

Every dynamic variable is given an index so we can retrieve it from the dynamic environment present in the stack. The variable `*dynamic-variables*` belongs to the realm of compilation so it is not essential to the virtual machine. Even so, it might be useful to establish error messages that report the name of a variable in case of anomalies.

We mentioned that we introduced dynamic variables by means of two special forms. To get them by means of functions would have been more faithful to the spirit of Scheme, but it would not have been equivalent. Since functions are invoked with computed arguments, the names of dynamic variables would have been computed, too, but that is not the case with special forms. Obviously, it would not have been possible to number dynamic variables if we had relied on functions to introduce them since any conceivable symbol (not to mention their values) would have been available. **[see p. 50]** In that case, we would then have had to arrange for another compilation, for example, by explicitly referring to symbols, which are afterall structures of considerable size that may pose problems in comparisons especially if we add packages or interning.

7.9 Exceptions

Every real language defines some way of handling errors. Error handling makes it possible to build robust, autonomous applications. There are, however, no specifications for error handling in Scheme, so we'll introduce error handling ourselves just to show what it is without much ado.

The idea of errors actually covers several different kinds in reality. First under this heading, we find unforeseen situations—situations we did not want but for which we can test. The underlying system may stumble, either because of problems with the type, `(car 33)`, or with the domain, `(quotient 111 0)`, or with arity `(cons)`. With appropriate predicates, we can test explicitly for those situations. However, other events can occur, such as an attempt to open a non-existing file. Handling that kind of error necessitates the following:

1. a way of reifying the error in a data structure that a user's programs can understand;
2. a call to the function that the user designates as the error handler.

Once this mechanism has been built into a language, users themselves want to exploit it. For that reason, errors take on the name *exceptions*, and thus is born *programming by exception* where the user programs only the right case and leaves the exiting (possibly even multiple exits) to the exception handler.

There are several models for exceptions. These models base practically all research about exception handlers on the idea of dynamic extent. When we want to protect a computation from the effects of exceptions, we associate a function for catching errors with the computation throughout its extent. If the computation

is completed without exception, then the error handler no longer has a purpose and becomes inactive. In contrast, if an error occurs, the error handler must be invoked. If the user invokes the exception system, it will provide the exception object. In contrast, if the system discovers the situation, then it will be up to the system to build the exception.

Among the models for exceptions, there are those with and without resumption. When certain exceptions are signaled (in particular, those that the programmer can signal, like **cerror** in COMMON LISP) it is sometimes possible to take up the computation again in the same place where it was interrupted. In many other cases, however, that's not feasible, and the only possible control operation is to escape to a safe context.

The exception model in COMMON LISP is more than complete, but its size is contrary to the design of this book, which is to show only the essentials. The model of ML does not support recovery; when an exception is signaled, the stack is unwound to the level it had when the exception handler was specified. It's possible to restart the exception to take advantage of the preceding handler. That model is not convenient for us because it loses the environment for dynamic variables that was present when the exception occurred. So here's the model we propose. (It's strongly inspired by the model of EULISP.) We'll first describe it informally and then implement it, hoping all the while that the two coincide!

The special form **monitor** associates a function for handling exceptions with the computation corresponding to its body. Its syntax is thus:

(**monitor** *handler forms* ...)

So we'll add the special form **monitor** to the syntactic analyzer **meaning**, like this:

```
... ((monitor) (meaning-monitor (cadr e) (cddr e) r tail?)) ...
```

The handler is evaluated and becomes the current exception handler. The forms in the body of **monitor** are consequently evaluated as if they were in a **begin** form. If no exception is signaled during the computation, the form **monitor** returns the value of the last form in its body and then reactivates the exception handler, which was hidden by the form **monitor**.

If an exception is signaled, then we search for the current handler, that is, the one associated by the dynamically closest **monitor**. (In implementation terms, we search for the handler highest in the evaluation stack.) Changing neither the stack nor its height nor its contents, we invoke the handler with two arguments: a Boolean indicating whether the exception can be continued and the object representing the exception. Since we have to foresee the possibility that the error handlers themselves may be erroneous, the current handler is executed under the control of the handler that it hides. The computation undertaken by the handler can either escape or return a value. If the exception could be continued, that value becomes the value expected in the place where the exception was signaled. If the exception could not be continued, then an exception is signaled—one that cannot be continued. When we exit from the handler by escaping, the computation is once again controlled by the nearest handler. Finally, there is a basic handler at the bottom of the stack. If it is ever invoked, it stops the program that is running and returns to the operating system (or to whatever takes the place of an operating

system), as in Figure 7.8.

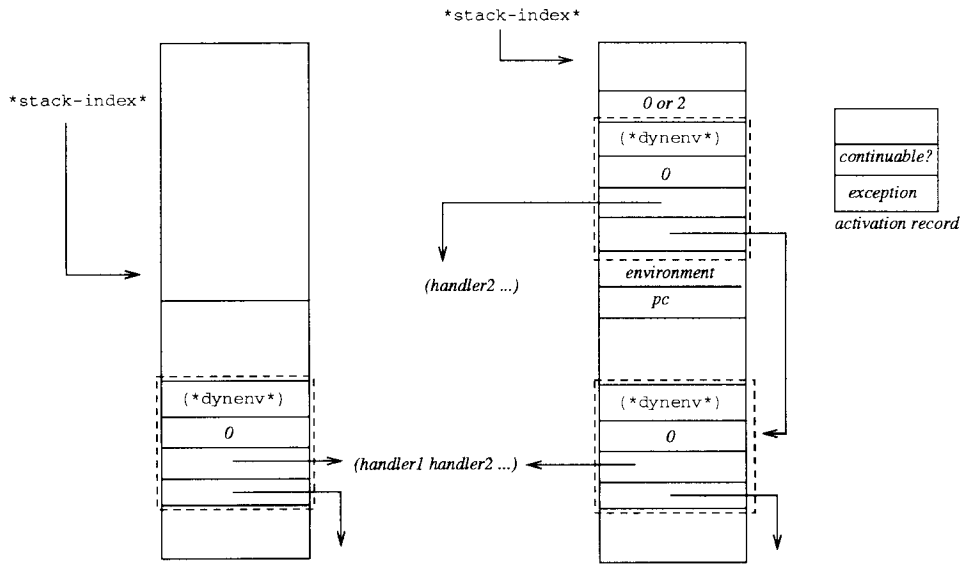


Figure 7.8 Signaling an exception

The pretreatment associated with the form `monitor` uses two new generators to manage the hierarchy of handlers.

```
(define (meaning-monitor e e+ r tail?)
  (let ((m (meaning e r #f))
        (m+ (meaning-sequence e+ r #f)) )
    (append m (PUSH-HANDLER) m+ (POP-HANDLER)) ) )
(define (PUSH-HANDLER) (list 246))
(define (POP-HANDLER)  (list 247))
```

To treat the case of exceptions that cannot be continued yet are continued anyway, we'll add the generator `NON-CONT-ERR`.

```
(define (NON-CONT-ERR) (list 245))
```

The instructions `PUSH-HANDLER` and `POP-HANDLER` just call the appropriate functions.

```
(define-instruction (PUSH-HANDLER) 246
  (push-exception-handler) )
(define-instruction (POP-HANDLER) 247
  (pop-exception-handler) )
```

Now we're ready to get to work. To determine which handler is closest to the top of the stack is easy. That's a task for dynamic binding, so we'll use dynamic binding. As a consequence, among other things, we won't have to create a new register containing the list of active handlers. For that reason, we will use the index 0 to store handlers because we realize that the index 0 can not have been attributed by the function `get-dynamic-variable-index`. The subtle point here

is that a handler should be invoked under the control of the preceding handler. We'll resolve that problem by associating the index 0 not with the handler itself but rather with the list of handlers, and make it responsible for putting the right list of handlers on the stack when an exception is signaled.

```
(define (search-exception-handlers)
  (find-dynamic-value 0) )
(define (push-exception-handler)
  (let ((handlers (search-exception-handlers)))
    (push-dynamic-binding 0 (cons *val* handlers)) ) )
(define (pop-exception-handler)
  (pop-dynamic-binding) )
```

When no exception is signaled, the handlers are pushed and popped without upsetting the evaluation discipline. Exceptions are signaled by the function `signal-exception`, which replaces the old `wrong` that we used to use.

```
(define (signal-exception continuable? exception)
  (let ((handlers (search-exception-handlers))
        (v* (allocate-activation-frame (+ 2 1))) )
    (set-activation-frame-argument! v* 0 continuable?)
    (set-activation-frame-argument! v* 1 exception)
    (set! *val* v*)
    (stack-push *pc*)
    (preserve-environment)
    (push-dynamic-binding 0 (if (null? (cdr handlers)) handlers
                                (cdr handlers) ))
    (if continuable?
        (stack-push 2)    ;pc for (POP-HANDLER) (RESTORE-ENV) (RETURN)
        (stack-push 0) ) ;pc for (NON-CONT-ERR)
    (invoke (car handlers) #t) ) )
```

The function `signal-exception` is called with two arguments: a Boolean that indicates whether or not the function must call the handler in a way that can be continued; and a value representing the exception. Any value is acceptable here, and predefined exceptions are encoded here as lists where the first term is a character string describing the anomaly. COMMON LISP and EULISP reify exceptions as real objects whose class is significant. First, we search for the list of active handlers; then we allocate an activation record and fill it in for the call to the first among those active handlers. The subtle part here is that we must prepare the stack, which is, after all, in an unknown state because we don't know yet where the exception occurred. We begin by saving the program counter and the environment (in case we need to return and restart there); then we save the list of handlers, shortened by the first one. Since there must be at least one handler in the list, we'll make sure that the program begins with a first (and last) handler, one that we never remove. Of course, that handler must never commit an error; we're sure of that since it sends an error message and returns control to the operating system. Now how do we distinguish an error that can be continued from one that cannot? One easy way is to assume that code in memory contains adequate instructions at fixed addresses. We've already put the instruction `(FINISH)` in a well known place. We can add others there, too.

```
(define (code-prologue)
  (set! finish-pc 1)
  (append (NON-CONT-ERR) (FINISH) (POP-HANDLER) (RESTORE-ENV) (RETURN)) )
```

Thus it's simple to make it possible to continue or not continue exception handling: that's represented by the address to return to, an address that was put onto the stack and that will be retrieved by **(RETURN)** which closes the handler. The address 0 leads to **(NON-CONT-ERR)** which indicates a new exception; the address 2 pops whatever **signal-exception** pushed onto the stack and returns to the place from which we left.

```
(define-instruction (NON-CONT-ERR) 245
  (signal-exception #f (list "Non continuable exception continued")) )
```

There's nothing left to do except show how to start the machine in spite of these additional constraints. We'll enrich the function **run-machine** so that it puts the ultimate handler in place. That ultimate handler could be programmed, for example, to print its final state and then stop. With no more ado, that gives us this:

```
(define (run-machine pc code constants global-names dynamics)
  (define base-error-handler-primitive
    (make-primitive base-error-handler) )
  (set! sg.current (make-vector (length global-names) undefined-value))
  (set! sg.current.names    global-names)
  (set! *constants*         constants)
  (set! *dynamic-variables* dynamics)
  (set! *code*              code)
  (set! *env*               sr.init)
  (set! *stack-index*       0)
  (set! *val*               'anything)
  (set! *fun*               'anything)
  (set! *arg1*              'anything)
  (set! *arg2*              'anything)
  (push-dynamic-binding 0 (list base-error-handler-primitive))
  (stack-push finish-pc)                ;pc for FINISH
  (set! *pc*                pc)
  (call/cc (lambda (exit)
             (set! *exit* exit)
             (run) )) )
(define (base-error-handler)
  (show-registers "Panic error: content of registers:")
  (wrong "Abort") )
```

The function **signal-exception** could be made available to programs. That function (like **error/cerror** in COMMON LISP or EULISP) signals and traps its own exceptions. However, its cost should limit it to exceptional situations.

In this section, we've defined an exception handler. This model makes it possible to return and restart after an exception. It also invokes the handler in the dynamic environment where the exception occurred, thus providing more possibilities as far as precisely detecting the context of the anomaly without affecting it. For example, with this model, we can write an unusual version of the factorial, like this:

```
(monitor (lambda (c e) ((dynamic foo) 1)
```

```
(let fact ((n 5))
  (if (= n 0) (/ 11 0)
      (* n (bind-exit (k)
             (dynamic-let (foo k)
               (fact (- n 1)) ) )) ) ) )
```

To avoid losing too much information about the context of the anomaly, it's a good idea not to escape too soon.

Since we're using a virtual machine where the code is represented by bytes, we've simplified the implementation of exception handling since exceptions can occur only during the treatment of instructions, and instructions are highly individualized, so the resources of the machine are always consistent between two instructions. That's not the case for a real implementation, which might receive asynchronous signals in unforeseen states.

You might ask why some predefined exceptions can be continued whereas others cannot. A byte machine is so regular that all exceptions could be made to continue because they can always be associated with a well defined program counter. Once again, that's not the case for a real implementation where either the idea of a program counter is a little vague or an anomaly, such as division by zero, cannot be detected until after the faulty operation. It is likely that only exceptions signaled by the user can be continued in a way that's portable.

Along these lines, it is pertinent to ask whether it's necessary to signal pre-treatment errors by `signal-exception` and to ask among other things whether these errors can be continued or not. Since the handler is consequently the responsibility of the compiler, we can envisage the compiler using exceptions that can be continued to correct certain anomalies on the fly.

7.10 Compiling Separately

As it is practiced in languages like C or Pascal, compilation happens through files. This section takes up that theme and presents such a compiler with an autonomous executable launcher. We'll also look at linking in this section.

7.10.1 Compiling a File

Compiling files poses hardly any problems. The function `compile-file` which follows here easily meets the challenge. It initializes the compilation variables `g.current` to represent the mutable global environment, `*quotation*` to collect quotations, and `*dynamic-variables*` to gather dynamic variables, of course. It also compiles the contents of the file, considering that like a huge `begin`, and saves the outcome (that is, the new values of those three variables along with the code) in the resulting file.

```
(define (read-file filename)
  (call-with-input-file filename
    (lambda (in)
      (let gather ((e (read in))
                   (content '()) )
        (if (eof-object? e)
```

```
            (reverse content)
            (gather (read in) (cons e content)) ) ) ) ) )
(define (compile-file filename)
  (set! g.current '())
  (set! *quotations* '())
  (set! *dynamic-variables* '())
  (let* ((complete-filename (string-append filename ".scm"))
         (e                 `(begin . ,(read-file complete-filename)))
         (code              (make-code-segment (meaning e r.init #t)))
         (global-names      (map car (reverse g.current)))
         (constants         (apply vector *quotations*))
         (dynamics          *dynamic-variables*)
         (ofilename         (string-append filename ".so")) )
    (write-result-file ofilename
                       (list ";;; Bytecode object file for "
                             complete-filename )
                       dynamics global-names constants code
                       (length (code-prologue)) ) ) )
(define (write-result-file ofilename comments
                           dynamics global-names constants code entry )
  (call-with-output-file ofilename
    (lambda (out)
      (for-each (lambda (comment) (display comment out))
                comments )(newline out)(newline out)
      (display ";;; Dynamic variables" out)(newline out)
      (write dynamics out)(newline out)(newline out)
      (display ";;; Global modifiable variables" out)(newline out)
      (write global-names out)(newline out)(newline out)
      (display ";;; Quotations" out)(newline out)
      (write constants out)(newline out)(newline out)
      (display ";;; Bytecode" out)(newline out)
      (write code out)(newline out)(newline out)
      (display ";;; Entry point" out)(newline out)
      (write entry out) (newline out) ) ) )
```

In order not to burden ourselves with problems tied to the file system and to the struture of file names, we'll assume (like in Scheme) that file names can be indicated by character strings. The compiler accepts such a string corresponding to the root of the file name. (By "root," we mean the name stripped of its usual suffix, here, `".scm"`.) The result of compilation will be stored in a file named by the same root suffixed by `".so"`. That result is made up of the code produced, the list of names of mutable global variables, the list of dynamic variables, the list of quotations, and the program counter for the first instruction to execute in the code. Always trying to simplify, we'll build the output file by means of `write` in its simplest style. We'll also add a few comments there to help a human read the result.

If the file to compile is this:

file si/example.scm

```
(set! fact
      ((lambda (fact) (lambda (n)
                        (if (< n 0)
                            "Toctoc la tete!"
                            (fact n fact (lambda (x) x)) ) ))
       (lambda (n f k)
         (if (= n 0)
             (k 1)
             (f (- n 1) f (lambda (r) (k (* n r)))) ) ) ) )
```

Then the result of compiling will be this:

file si/example.so

```
;;; Bytecode object file for si/example.scm

;;; Dynamic variables
()

;;; Global modifiable variables
(FACT)

;;; Quotations
#("Toctoc la tete!")

;;; Bytecode
#(245 20 247 38 43 40 30 59 74 32 1 34 81 35 106 31 10 3 34 82 34 51 60
  39 46 30 39 2 34 1 34 82 35 105 34 2 34 40 30 19 72 32 6 1 2 34 6 1 0
  34 1 35 109 34 51 60 39 46 43 34 53 62 61 60 39 46 43 34 51 60 32 40
  30 38 72 32 1 34 81 35 107 31 4 9 0 30 24 6 1 0 34 1 34 6 1 0 34 40
  30 4 72 32 1 43 34 53 62 61 60 39 46 43 33 27 0 43)

;;; Entry point
5
```

7.10.2 Building an Application

Compiling files is all well and good, but eventually we want to execute them! The second utility we'll add corresponds to what we often call a *linker* (the `ld` of UN⋆X). It organizes compiled files into a unique executable file. The function `build-application` takes the name of a file to generate and then the names of files to link.

```
(define (build-application application-name ofilename . ofilenames)
  (set! sg.current.names    '())
```

```
(set! *dynamic-variables* '())
(set! sg.current          (vector))
(set! *constants*         (vector))
(set! *code*              (vector))
(let install ((filenames (cons ofilename ofilenames))
              (entry-points '()) )
  (if (pair? filenames)
      (let ((ep (install-object-file! (car filenames))))
        (install (cdr filenames) (cons ep entry-points)) )
      (write-result-file application-name
                         (cons ";;; Bytecode application containing "
                               (cons ofilename ofilenames) )
                         *dynamic-variables*
                         sg.current.names
                         *constants*
                         *code*
                         entry-points ) ) ) )
```

Most of the work is carried out by **install-object-file!**, which installs a compiled file inside the five variables governing the machine:

- **sg.current.names** indicates the mutable global variables;
- ***dynamic-variables*** indicates the dynamic variables;
- **sg.current** contains the values of mutable global variables;
- ***constants*** contains the quotations;
- ***code*** is the byte vector of instructions.

After installing all these files, we only have to write the resulting executable file in a form that resembles the compiled files except for the entry point; it becomes a list of entry points.

The function **install-object-file!** installs a compiled file and returns the address of its first instruction. Putting the code of the file to install into the ***code*** vector is easy. What's harder is to make the files share what they have in common and to protect what each file has for its own use. That will be the purpose of the functions whose names begin with **relocate** in the following definition:

```
(define (install-object-file! filename)
  (let ((ofilename (string-append filename ".so")))
    (if (probe-file ofilename)
        (call-with-input-file ofilename
          (lambda (in)
            (let* ((dynamics     (read in))
                   (global-names (read in))
                   (constants    (read in))
                   (code         (read in))
                   (entry        (read in)) )
              (close-input-port in)
              (relocate-globals! code global-names)
              (relocate-constants! code constants)
              (relocate-dynamics! code dynamics)
              (+ entry (install-code! code)) ) ) )
        (signal #f (list "No such file" ofilename)) ) ) )
```

```
(define (install-code! code)
  (let ((start (vector-length *code*)))
    (set! *code* (vector-append *code* code))
    start ) )
```

Quotations, for example, belong privately to each file where they appear, and one should not be confused with another. In each file, then, they are numbered starting from zero and thus share the same numbers. Quotations will be concatenated in the variable ***constants***, and numbers referring to them will be updated. Those numbers appear in the vector of instructions as arguments to the instruction **CONSTANT**, so it is there that we must update them. To do so, we examine the vector of code part by part sequentially, like this:

```
(define CONSTANT-code 9)
(define (relocate-constants! code constants)
  (define n (vector-length *constants*))
  (let ((code-size (vector-length code)))
    (let scan ((pc 0))
      (when (< pc code-size)
        (let ((instr (vector-ref code pc)))
          (when (= instr CONSTANT-code)
            (let* ((i (vector-ref code (+ pc 1)))
                   (quotation (vector-ref constants i)) )
              (vector-set! code (+ pc 1) (+ n i)) ) )
          (scan (+ pc (instruction-size code pc))) ) ) ) )
  (set! *constants* (vector-append *constants* constants)) )
```

For global variables, in contrast, we must make sure they are shared. Any two files that both use **foo** must share that variable. Variables are numbered with respect to a local list of names. The only places where these numbers appear are as arguments to the instructions **GLOBAL-REF**, **CHECKED-GLOBAL-REF**, and **SET-GLOBAL!**. Each number we find will be associated with its external name; that name is associated with a number belonging to the variable in the executable; the executable number will eventually replace the other number.

```
(define CHECKED-GLOBAL-REF-code 8)
(define GLOBAL-REF-code 7)
(define SET-GLOBAL!-code 27)
(define (relocate-globals! code global-names)
  (define (get-index name)
    (let ((where (memq name sg.current.names)))
      (if where (- (length where) 1)
          (begin (set! sg.current.names (cons name sg.current.names))
                 (get-index name) ) ) ) )
  (let ((code-size (vector-length code)))
    (let scan ((pc 0))
      (when (< pc code-size)
        (let ((instr (vector-ref code pc)))
          (when (or (= instr CHECKED-GLOBAL-REF-code)
                    (= instr GLOBAL-REF-code)
                    (= instr SET-GLOBAL!-code) )
            (let* ((i (vector-ref code (+ pc 1)))
```

```
                    (name (list-ref global-names i)) )
                (vector-set! code (+ pc 1) (get-index name)) ) )
            (scan (+ pc (instruction-size code pc))) ) ) ) )
  (let ((v (make-vector (length sg.current.names) undefined-value)))
    (vector-copy! sg.current v 0 (vector-length sg.current))
    (set! sg.current v) ) )
```

The process is similar for dynamic variables.

```
(define DYNAMIC-REF-code 240)
(define DYNAMIC-PUSH-code 242)
(define (relocate-dynamics! code dynamics)
  (for-each get-dynamic-variable-index dynamics)
  (let ((dynamics (reverse! dynamics))
        (code-size (vector-length code)) )
    (let scan ((pc 0))
      (when (< pc code-size)
        (let ((instr (vector-ref code pc)))
          (when (or (= instr DYNAMIC-REF-code)
                    (= instr DYNAMIC-PUSH-code) )
            (let* ((i (vector-ref code (+ pc 1)))
                   (name (list-ref dynamics (- i 1))) )
              (vector-set! code (+ pc 1)
                           (get-dynamic-variable-index name) ) ) )
          (scan (+ pc (instruction-size code pc))) ) ) ) ) ) )
```

Notice that these functions use the function `instruction-size`. We should also note that examining the vector of code three times is a waste of effort; we should factor that work into a single pass.

The form we adopted for compiled files makes it easy to imagine new modes for combining files. The list of mutable global variables serves as a sort of interface to a file considered as a module. That module then exports all its mutable global variables under the name they had within the file. We could simply rename these variables or restrict them. We could even invent a language for linking that specifies how to group modules and manage the names of variables that they use. Let's explain a bit more about that language, inspired by the language proposed for modules of EuLisp in [QP91a]. As an example, the following directive defines what the module `mod` imports:

```
(ordered-union
   (only (fact) (expose "fact"))
   (union (except-pattern ("fib*") (expose "fib"))
          (rename ((call/cc call-with-current-continuation))
            (expose "scheme") )
          (expose "numeric") ) )
```

Let's assume that the notation `foo@mod` designates the variable named `foo` in the module `mod`. Let's also suppose that that the module `fact` defines the variables `fact` and `fact100` (containing the precalculated value of `(fact 100)`, a value often in demand); that the module `fib` defines the variables `fib`, `fib20`, and `Fibonacci`. The module `numeric` procures functions like `fact` and `fib`, while `scheme` procures all the functions of R4RS. The module produced by that directive is thus formed this way: it contains the variable `fact@fact`; (the variable `fact20@fact`, although

exposed by the directive (**expose "fact"**), is excluded by the restrictive directive **only**;) the sole variable **Fibonacci@fib**; (the other variables from the same module are excluded by the clause **except-pattern**;). The variable **call/cc@scheme** renames the variable **call-with-current-continuation@scheme**. Since the union creating **mod** is specified as ordered, the module **fact** will be evaluated before the others (**fib**, **numeric**, and **scheme**) whose order is not specified but must be compatible with their definition. Since the module **numeric** very probably uses the module **scheme**, it should be evaluated afterwards.

7.10.3 Executing an Application

The purpose of an executable is, of course, to be executed. Since we've prepared everything in advance, the execution becomes simple even if it is long and tedious as far as initializing all the registers.

```
(define (run-application stack-size filename)
  (if (probe-file filename)
      (call-with-input-file filename
        (lambda (in)
          (let* ((dynamics     (read in))
                 (global-names (read in))
                 (constants    (read in))
                 (code         (read in))
                 (entry-points (read in)) )
            (close-input-port in)
            (set! sg.current.names    global-names)
            (set! *dynamic-variables* dynamics)
            (set! sg.current (make-vector (length sg.current.names)
                                          undefined-value ))
            (set! *constants*         constants)
            (set! *code*              (vector))
            (install-code! code)
            (set! *env*               sr.init)
            (set! *stack*             (make-vector stack-size))
            (set! *stack-index*       0)
            (set! *val*               'anything)
            (set! *fun*               'anything)
            (set! *arg1*              'anything)
            (set! *arg2*              'anything)
            (push-dynamic-binding
             0 (list (make-primitive (lambda ()
                                       (show-exception)
                                       (*exit* 'aborted) ))) )
            (stack-push 1)                        ;pc for FINISH
            (if (pair? entry-points)
                (for-each stack-push entry-points)
                (stack-push entry-points) ) )
          (set! *pc* (stack-pop))
          (call/cc (lambda (exit)
                     (set! *exit* exit)
                     (run) )) ) ) )
```

```
(static-wrong "No such file" filename) ) )
```

We begin by reading the file containing the executable, and we close the input port as soon as it become useless. We initialize the global variables of the machine, like the function **run-machine** used to do. After reserving a place for the base exception handler, we push all the entry points for the compiled files participating in the executable. To be more general, we also consider a simple compiled file as an executable on its own. The default handler terminates execution if it is invoked; to do so, it captures the call continuation of **run-application** (the operating system) and then places it in the variable ***exit***. The function **(show-exception)** is responsible for printing a meaningful error message[12] on the basis of the exception present in the register ***val***. The only remaining task is to simulate a **(RETURN)** to evaluate the first file of the executable; that file will return to the second file, and so on down the line.

The function **run-application** takes the size of the evaluation stack to use as an argument. To keep the size of the stack within limits is a subtle but very important implementation point. It would be too costly to test whether we have overrun the stack at every **stack-push**. Sometimes, we could use the operating system or the properties of segmented memory to eliminate that test and increase the size of the stack in case of overrunning it. Here our simulation is particularly inefficient because we can never say often enough how expensive the form **(vector-ref** v i**)** is since it must verify that v is a vector and i is an positive integer less than the size of the vector.

The function **run-application** doesn't need much to execute a compiled file. It really needs only **run** and functions related to vectors, lists, and other classes like **primitive**, **continuation**, etc. It also needs a **read** function for quotations. Thus it's quite independent of the compiler that produced the application, (but it doesn't help us much with debugging the program) and that, of course, was the goal!

7.11 Conclusions

To get a real implementation of Scheme, we would have to add many missing functions as well as the corresponding data types. That presents hardly any problems except detecting errors in data types or domain.

The implementation we wrote earlier is general enough and certainly can be improved. It's interesting because it derives from the preceding efficient interpreter, the one that we reconfigured as a compiler. There is, in fact, a profound connection between an interpreter and compiler (explored in [Nei84]). An interpreter executes a program whereas a compiler transcribes a program as something that will be executed. Thus there is a simple difference in library—execution library or generation library—in question here. We've exploited that connection to derive the compiler from the interpreter.

However, we've inherited only the information that appears in the intermediate language, and that information is insufficient to insure a good compilation. We know nothing, for example, about the use of local variables; indeed that is the

12. For example, **"bus error; core not dumped."**

principal analysis that we lack. We don't know whether these variables can be modified, whether they are closed, multiply closed, modified while closed, etc. In many cases, all those static properties make it possible to get rid of activation records and to use only the stack—thus improving allocation and speed. Activation records are useful only to save the values of closed variables, so it's not necessary to allocate such records when variables are not closed. Since their values are already on the stack, as long as we leave them there and search for them there, we'll speed things up a lot.

7.12 Exercises

Exercise 7.1 : Modify the compiler defined in this chapter to use a register `*dynenv*` to indicate the dynamic environment. **[see p. 252]**

Exercise 7.2 : Define a function `load` to load a compiled program at execution and execute the program. For example, if `fact.scm` is a file compiled as `fact.so`, we should be able to compile and execute the following file:

```
(begin (load "fact")
       (fact 5) )
```

Exercise 7.3 : Write a function `global-value` to take the name of a global variable and return its value.

Exercise 7.4 : Modify the instructions involved with dynamic variables to implement them by shallow binding. **[see p. 23]**

Exercise 7.5 : **[see p. 265]** Write a function to rename exported variables. If `fact.so` is a compiled file, then the following lines should create a new module, `nfact.so` where the variable `fact` has been renamed as `factorial`.

```
(build-application-renaming-variables
  "nfact.so" "fact.so" '((fact factorial)) )
```

Exercise 7.6 : **[see p. 195]** Modify the instruction `CHECKED-GLOBAL-REF` so it can modify itself into `GLOBAL-REF` once the variable being read has been initialized.

Project 7.7 : Gnu Emacs Lisp in [LLSt93] and xscheme in [Bet91], among other implementations of Lisp or Scheme, have byte-code compilers. Adapt the compiler from this chapter to interpret byte-code implemented by those virtual machines.

Recommended Reading

There are few works that explain the rudiments of compilation. In fact, that's one reason for this book. Nevertheless, you might consult [All78] and [Hen80].

For higher order compilation, see [Dil88]. These sources also contain interesting compilers with comments: [Ste78, AS85, FWH92].

8
Evaluation & Reflection

UNIQUELY characteristic of Lisp is its evaluation mechanism: `eval`. Although this book talks relentlessly about evaluation, we haven't said a word yet about the problem of making evaluation available to programmers. Evaluation poses a number of problems with respect to specification, integrity, and linguistics. Some people are thinking of all these problems when they say concisely, "`eval` is *evil*." Catching its genius in a useful form is the first step toward programming reflection, a topic this chapter also covers.

For 271 pages now, we've been presenting various interpreters detailing the core of the evaluation mechanism. For most of them, making the evaluation mechanism accessible to programmers is trivial, a task requiring very little code. That's what implementers have been doing for ages. The existence of such a mechanism [**see p. 2**] was surely one of the goals in creating Lisp. From the very beginning of the sixties, in fact, making the `eval` function explicit showed up in the writings of the founders, such as [McC60, MAE+62].

Explicit evaluation is fundamental, supporting as it does so many effects, notably, a powerful system of macros, immersion of the programming environment within the language, and pronounced reflection. Of course, explicit evaluation also has some defects such as macros, a programming environment right inside the language, and invasive reflection. Like a magic djinn, explicit evaluation can be both useful and dangerous.

What sort of contract should explicit evaluation satisfy? Clearly, we would like to say that:

$$\texttt{(eval '}\pi\texttt{)} \quad \equiv \quad \pi \tag{1}$$

But right now we're going to show you the ambiguity of that formula.

8.1 Programs and Values

Let's actually try to put `eval` inside the first interpreter in this book. [**see p. 3**] That interpreter was written in pure Scheme, without any particular restrictions. Let's assume first that `eval` should be a special form. In consequence, it will appear in `evaluate` which then becomes this:

```
(define (evaluate e env)
```

```
(if (atom? e)
    (cond ((symbol? e) (lookup e env))
          ((or (number? e)(string? e)(char? e)(boolean? e)) e)
          (else (wrong "Cannot evaluate" e)) )
    (case (car e)
      ((quote)  (cadr e))
      ((if)     (if (not (eq? (evaluate (cadr e) env)
                              the-false-value ))
                    (evaluate (caddr e) env)
                    (evaluate (cadddr e) env) ))
      ((begin)  (eprogn (cdr e) env))
      ((set!)   (update! (cadr e) env (evaluate (caddr e) env)))
      ((lambda) (make-function (cadr e) (cddr e) env))
      ((eval)   (evaluate (evaluate (cadr e) env) env)) ;** Modified **
      (else     (invoke (evaluate (car e) env)
                        (evlis (cdr e) env) )) ) ) )
```

As a special form, **eval** begins by evaluating the form that corresponds to its first argument (just like a function does); then it evaluates the resulting value in the current environment. This patently trivial description of what it does hides some gaping questions.

As we've emphasized time and again, the function **evaluate** entails a first stage of syntactic analysis and a second stage of evaluation. We separated those stages into **meaning** and **run** in the most recent interpreters. The first argument of **evaluate** corresponds to a program, whereas its result belongs to the domain of values. There's a problem, then, (let's call it a type-checking problem) with the form (**evaluate** (**evaluate** ...) ...) where the outermost **evaluate** is applied to a value and not to a program. Are programs values? Are values programs?

Programming languages are usually defined by a grammar specifying their syntactic form. The grammar of Scheme appears in [CR91b]. It specifies that certain arrangements of parentheses and letters are syntactically legal programs. The grammar also specifies the syntax of data, and we can check that any program conforms to that syntax for data. The **read** function, in fact, is universal in the sense that it can read both programs and data. However, nothing makes it obligatory for programs to be read by **read** in order to be evaluated, even if doing so is simpler. A Smalltalk evaluator, as in [GR83], for example, reads its programs in windows with a special reader that stores positions in terms of rows and columns in order to highlight portions of these programs that are syntactically incorrect. Since this usage prevails, of course, it is clear that any program respecting the grammar of Scheme is or can be associated with a legitimate value according to this same grammar.

Conversely and no less clear, there are many values that correspond to programs but even more that don't, for example, (**quote** . 1). Unfortunately, there remain values whose status is not so clear.

- Take a value more or less resembling a program apart from a few clinkers, for example, (**if #t 1** (**quote** . 2)). Is it a program? That expression poses no problem for the operational definition we just gave for **eval** since (**quote** . 2) is not evaluated, but this expression hardly conforms to the grammar of Scheme programs.

- Take a value such as the one constructed by the expression `'(',car '(a b))` and having in its function position a form quoting the value of the function `car`. Is it a program? That value is not a program according to the syntax of R^{4}RS because it has no external representation: it cannot be typed with a normal keyboard. Yet once again, it poses no problem for `eval` as we described it before.
- Let's consider the syntactic conventions of COMMON LISP where `#1=` is a name for the expression that follows it, and `#1#` represents the expression of name `1`. With that in mind, let's consider the following expression, whose graphic representation is shown in Figure 8.1.

```
(let ((n 4))
  #1=(if (= n 1) 1
         (* n ((lambda (n) #1#) (- n 1))) ) )
```

This value has a cycle. In fact, we would say that the program involved is syntactically recursive, so it is not syntactically legal in Scheme. Once again, it poses no problem, however, for the `eval` form that we described earlier.

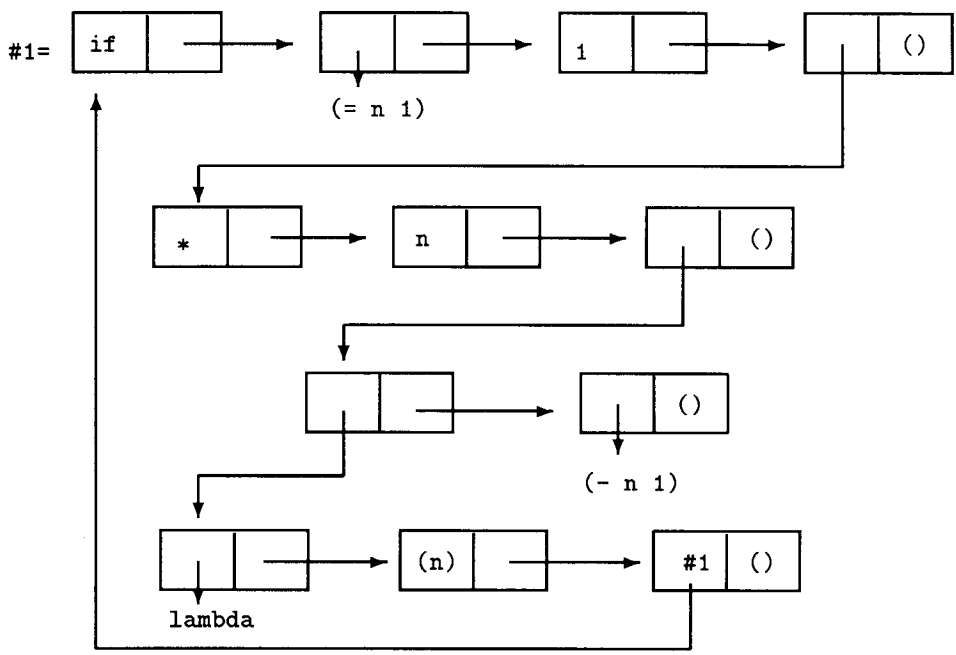


Figure 8.1 Syntactically recursive factorial

From the grammatical point of view, those expressions all look like they come from a careful study in how to produce monstrosities, but they clearly show that the idea of a program is subtle with respect to `eval`. The same problem arises about macros that carry out computations on representations of programs. Here again we find the distinction between static and dynamic errors that we discussed

earlier. [see p. 194] Drawing the line is difficult, but we'll risk doing so! The most permissive hypothesis is to allow everything, so a syntactic anomaly like `(quote . 3)` is not detected until we're obliged to evaluate the program. The strictest hypothesis is to accept only those programs represented by finite acyclic graphs where all nodes are syntactically correct. That rule eliminates the syntactically recursive factorial—the one we offered as an example just before—even though it is syntactically correct. This hard line—the one taken by Scheme—is the one we'll take since it has been shown in [Que92a] that syntactically recursive programs can be rewritten purely as trees with no cycles. As for quotations, we'll authorize any finite value for which the atoms (the leaves of the tree) have an external representation. This rule excludes quotations that entail closures, continuations, or streams, as well as cyclic[1] structures.

Operationally, we'll use the following predicates to characterize the values that we allow as legitimate programs. These predicates test syntax and detect[2] cycles.

```
(define (program? e)
  (define (program? e m)
    (if (atom? e)
        (or (symbol? e) (number? e) (string? e) (char? e) (boolean? e))
        (if (memq e m) #f
            (let ((m (cons e m)))
              (define (all-programs? e+ m+)
                (if (memq e+ m+) #f
                    (let ((m+ (cons e+ m+)))
                      (and (pair? e+)
                           (program? (car e+) m)
                           (or (null? (cdr e+))
                               (all-programs? (cdr e+) m+) ) ) ) ) )
              (case (car e)
                ((quote) (and (pair? (cdr e))
                              (quotation? (cadr e))
                              (null? (cddr e)) ))
                ((if) (and (pair? (cdr e))
                           (program? (cadr e) m)
                           (pair? (cddr e))
                           (program? (caddr e) m)
                           (pair? (cdddr e))
                           (program? (cadddr e) m)
                           (null? (cddddr e)) ))
                ((begin) (all-programs? (cdr e) '()))
                ((set!) (and (pair? (cdr e))
                             (symbol? (cadr e))
                             (pair? (cddr e))
                             (program? (caddr e) m)
                             (null? (cdddr e)) ))
                ((lambda) (and (pair? (cdr e))
```

1. Allowing them is not too costly, and they may be useful, for example, in the implementation of MEROONET.

2. Note that the value `(let ((e '(x))) '(lambda ,e ,e))` is a legitimate program.

```
                                (variables-list? (cadr e))
                                (all-programs? (cddr e) '()) ))
                  (else (all-programs? e '())) ) ) ) ) )
  (program? e '()) )
(define (variables-list? v*)
  (define (variables-list? v* already-seen)
    (or (null? v*)
        (and (symbol? v*) (not (memq v* already-seen)))
        (and (pair? v*)
             (symbol? (car v*))
             (not (memq (car v*) already-seen))
             (variables-list? (cdr v*)
                              (cons (car v*) already-seen) ) ) ) )
  (variables-list? v* '()) )
(define (quotation? e)
  (define (quotation? e m)
    (if (memq e m) #f
        (let ((m (cons e m)))
          (or (null? e)(symbol? e)(number? e)
              (string? e)(char? e)(boolean? e)
              (and (vector? e)
                   (let loop ((i 0))
                     (or (>= i (vector-length e))
                         (and (quotation? (vector-ref e i) m)
                              (loop (+ i 1)) ) ) ) )
              (and (pair? e)
                   (quotation? (car e) m)
                   (quotation? (cdr e) m) ) ) ) ) )
  (quotation? e '()) )
```

Armed with the predicate **program?**, we can improve our formulation of the special form **eval** in this way:

```
... ((eval) (let ((v (evaluate (cadr e) env)))
              (if (program? v)
                  (evaluate v env)
                  (wrong "Illegal program" v) ) ))  ...
```

Alas, that form is still clumsy because the preceding predicates do not test whether a value *is* a program and a quotation *is* correct; they only test whether a value is a *legitimate representation* of a program or quotation. That's a pledge of the intelligibility of a value in a particular role. The value is neither the program itself nor the quotation. Indeed, a program and a quotation are whatever the interpreter needs for them to be. To refine these ideas, let's look now at the pre-denotational interpreter of Chapter 4. [see p. 111] There memory and continuations were explicit, and inside them the data of interpreted Scheme were represented by closures simulating objects. Here's that interpreter, but we've added the special form **eval**:

```
(define (evaluate e r s k)
  (if (atom? e)
      (if (symbol? e) (evaluate-variable e r s k)
          (evaluate-quote e r s k) )
      (case (car e)
```

```
        ((quote)  (evaluate-quote (cadr e) r s k))
        ((if)     (evaluate-if (cadr e) (caddr e) (cadddr e) r s k))
        ((begin)  (evaluate-begin (cdr e) r s k))
        ((set!)   (evaluate-set! (cadr e) (caddr e) r s k))
        ((lambda) (evaluate-lambda (cadr e) (cddr e) r s k))
        ((eval)   (evaluate-eval (cadr e) r s k))          ;** Modified **
        (else     (evaluate-application (car e) (cdr e) r s k)) ) ) )
(define (evaluate-eval e r s k)
  (evaluate e r s
    (lambda (v ss)
      (let ((ee (transcode-back v ss)))
        (if (program? ee)
            (evaluate ee r ss k)
            (wrong "Illegal program" ee) ) ) ) ) )
```

In this interpreter, the value we get by evaluating the first argument of the **eval** form is first decoded as its external form (by **transcode-back**). Then the nature of the program is checked so that it is eventually evaluated. The variables **e** and **ee** have as their domain the descriptions of programs, while the variable **v** indicates the values. The role of **transcode-back** is to take a value and put it into such a form that it can be considered as a description of the program. This is just what gets the effect of **eval** in a Scheme, where there is no **load** function available: write the value to evaluate into a file; use **display** for **transcode-back**; then evaluate that file by **load** instead of **evaluate**. In that case, the program is "compiled"[3] in the name of the file.

Let's take a last example, this time, the fast compiling interpreter from Chapter 6. **[see p. 183]** It reveals a few more interesting points. There, values and descriptions of programs share the same support; thus it will not be necessary to introduce a conversion between them. In contrast, the goal of the fast interpreter was to suppress evaluation for any static calculation that could be carried out before hand.

```
(define (meaning e r tail?)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e r tail?)
                      (meaning-quotation e r tail?) )
      (case (car e)
        ((quote)  (meaning-quotation (cadr e) r tail?))
        ((lambda) (meaning-abstraction (cadr e) (cddr e) r tail?))
        ((if)     (meaning-alternative (cadr e) (caddr e) (cadddr e)
                                       r tail? ))
        ((begin)  (meaning-sequence (cdr e) r tail?))
        ((set!)   (meaning-assignment (cadr e) (caddr e) r tail?))
        ((eval)   (meaning-eval (cadr e) r tail?))         ;** Modified **
        (else     (meaning-application (car e) (cdr e) r tail?)) ) ) )
(define (meaning-eval e r tail?)
  (let ((m (meaning e r #f)))
    (lambda ()
      (let ((v (m)))
```

3. In fact, coding **eval** by **load** provides an **eval** only at *toplevel*. We'll get to that idea later.

```
(if (program? v)
    (let ((mm (meaning v r tail?)))
      (mm) )
    (wrong "Illegal program" v) ) ) ) ) )
```

Now it becomes clear that for this fast compiling interpreter, which converts a description of a program into a tree of thunks, that compiling the value **v** cannot be done statically. For the first time, a thunk will be generated which does not contain only simple calculations like allocations, conditionals, sequences, or read-access or write-access inside various data structures. Instead, (oh, horrors!) it entails a call to **meaning**. This call thus implies that we must keep the code for **meaning** and its affiliates at execution: not a slight cost!

We've tried here to eliminate the confusion between values and programs. This confusion is on the same order as that between quotations and values. There's a reason that **eval** and **quote** are often said to play reverse roles: for its argument, **quote** takes a description of a value to synthesize whereas **eval** starts from a value and converts it into a program to evaluate. Both perform conversions between signified and signifier, so it is important not to confuse them.

8.2 eval as a Special Form

When **eval** is a special form, we get close to the ideal that we advocated in equation (1); that is, we make (**eval** (**quote** π)) equivalent to π. That's just what we want if we adopt equational thinking, as in [FH89, Mul92]. In a well built theory, we can substitute two things that are mutually equal *in any context.* More precisely, that implies that (1) is an attenuated form of (2), where $C[\,]$ represents a context.

$$\forall C[\,],\quad C[\texttt{(eval '}\pi\texttt{)}] \;\equiv\; C[\pi] \tag{2}$$

Accordingly, the evaluation of a closed form (without free variables) like (**eval** '((**lambda** (**x y**) **x**) **1 2**)) returns **1**. Moreover, we can also take advantage of the current lexical environment, like this:

```
((lambda (x) (eval 'x))
 3 )                          → 3
((lambda (x y) (eval y))
 4 'x )                       → 4
((lambda (x y z) (eval y))
 5 (list 'eval 'z) 'x )       → 5
```

This effect is made possible only because the current lexical environment is available. About the fast compiling interpreter, we can even observe that it is not only necessary to have the compiler **meaning** in a working state at execution time, but it is also necessary to capture the current lexical environment in order to take up the compilation again in the same environment. For that reason the generated thunk encloses not only **meaning** but also **r** (and as an accessory, **tail?**): all of them must be present at execution.

This effect is even more pertinent if we show what has become of the byte-code compiler from Chapter 7. [see p. 223] The analyzing function **meaning-eval** calls the byte-code generator **EVAL/CE** (for *eval in the current environment*). This

generator receives the argument to evaluate as well as the lexical environment for compilation, like this:

```
(define (meaning-eval e r tail?)
  (let ((m (meaning e r #f)))
    (EVAL/CE m r) ) )
(define (EVAL/CE m r)
  (append (PRESERVE-ENV) (CONSTANT r) (PUSH-VALUE)
          m (COMPILE-RUN) (RESTORE-ENV) ) )
```

The byte-code generator **EVAL/CE** systematically preserves the execution environment because we don't take account of the parameter `tail?`. A new intruction **(COMPILE-RUN)** condenses the whole compiler-evaluator. That instruction takes the expression to evaluate in the register ***val*** and the compilation environment on the top of the stack and delegates to **compile-on-the-fly** the task of compiling the program and installing its code in memory to execute as if it were the body of a function.

```
(define (COMPILE-RUN) (list 255))

(define-instruction (COMPILE-RUN) 255
  (let ((v *val*)
        (r (stack-pop)) )
    (if (program? v)
        (compile-and-run v r #f)
        (signal-exception #t (list "Illegal program" v)) ) ) )

(define (compile-and-run v r tail?)
  (unless tail? (stack-push *pc*))
  (set! *pc* (compile-on-the-fly v r)) )
(define (compile-on-the-fly v r)
  (set! g.current '())
  (for-each g.current-extend! sg.current.names)
  (set! *quotations* (vector->list *constants*))
  (set! *dynamic-variables* *dynamic-variables*)
  (let ((code (apply vector (append (meaning v r #f) (RETURN)))))
    (set! sg.current.names (map car (reverse g.current)))
    (let ((v (make-vector (length sg.current.names) undefined-value)))
      (vector-copy! sg.current v 0 (vector-length sg.current))
      (set! sg.current v) )
    (set! *constants* (apply vector *quotations*))
    (set! *dynamic-variables* *dynamic-variables*)
    (install-code! code) ) )
```

Compiling on the fly means that we have to update the global variables for compilation: **g.current** (the environment of mutable global variables), ***quotations***, and ***dynamic-variables***.[4] The compiled code is followed by a **(RETURN)** so that it can return to its caller. After the compilation, we enrich the execution environment in order to add new quotations, mutable global variables, or dynamic

4. An contrast to other variables, ***dynamic-variables*** is shared between the compiler and the executing machine. Also idempotent assignments to it are pointless, but those assignments indicate what must change if they were not shared.

variables created at that time. There's nothing left to do but to install the new code segment and then execute it.

This installation is analogous to dynamic linking. It shows that for a language like C, `eval` comes down to writing a character string in a file, then compiling that file by a call to `cc`, then loading that file by dynamic linking, and finally executing it. Code installed that way can't be eliminated in the current state of the machine and thus in the long term it will obstruct memory. Recuperating memory consumed by compiled code is always a touchy issue.

The cost of introducing this new special form is thus

- a new instruction (`COMPILE-RUN`) accompanied by the function **meaning** and a few other utilities, representing a great increase in size for small applications;
- and, at every call to `eval`, supplementary quotations to store the necessary compilation environments.

Fortunately, we have to save these compilation environments only in these places. The marginal cost is thus proportional to the number of `eval` forms.

As far as implementation and debugging, inserting an `eval` form makes it possible to program a local interaction loop to check or modify local variables by their own name. If a debugger were available and could be activated by a simple interruption (by an interruption from the keyboard, for example) at any stage of the program where we wanted, it would require us to keep the entire text of the program as well as saving all the compilation environments.

8.3 Creating Global Variables

Because of the explicit contract in equation (2), the expression `(eval '(set! foo 33))` should be equivalent to `(set! foo 33)`. We've already seen its various semantics [see p. 111] in Chapter 4. They were analyzed in Section 6.1.9 [see p. 203]. If simply mentioning the name of a variable makes it exist, then `eval` should behave likewise. In contrast, if variables must be declared before they are used, then `eval` should conform to that practice instead. The function `compile-on-the-fly` extends the global evaluation environment on its return in order to take into account new mutable global variables that have just appeared.

This point of that remark is to make you realize that equations (1) or (2) stipulate that not only the values of π and `(eval 'π)` but also their induced effects (such as any global variables created, any modifications of the global environment or the evaluation context) should be the same. This idea of "same" is taken into account by equation (2) which must remain valid regardless of the context. That means that π and `(eval 'π)` must be indistinguishable. In other words, there can be no program[5] that we can write that can tell them apart.

5. This statement is somewhat theoretical since we might let the program check the clock to measure its own execution time and by that means, a program could deduce whether it had encountered an `eval` or not.

8.4 eval as a Function

The special form **eval** evaluates its argument as a function would do, so we might as well ask ourselves whether we could achieve the same evaluation as a function rather than a special form. To that end, let's take each of the various interpreters again and show how to introduce such a function in each one.

With the naive interpreter at the beginning of this book [see p. 3], we would write this:

```
(defprimitive eval
  (lambda (v)
    (if (program? v)
        (evaluate v env.global)
        (wrong "Illegal program" v) ))
  1 )
```

Right away, the major difference is apparent: we've lost the current lexical environment. Since we still have to give one to the evaluator, **evaluate**, we'll provide the only one that's visible: the global environment! As a consequence, we have here a function playing the role of a global evaluator equivalent to the one used by the toplevel loop. We'll name it **eval/at** for *eval at top-level* while we'll name the preceding special form **eval/ce** to distinguish it when we feel the need to do so. Using **eval/at** is not a complete loss because we save some effort with it: we no longer have to store the lexical environments present when **eval/ce** is called. We save a few quotations that way.

To clarify these ideas, we'll take up the preceding examples again: assuming that **eval/at** as a function leads to this:

```
(set! x 2)(set! z 1)
((lambda (x) (eval/at 'x))
 3 )                              → 2
((lambda (x y) (eval/at y))
 4 'x )                           → 2
((lambda (x y z) (eval/at y))
 5 (list 'eval/at 'z) 'x )        → 1
```

Even if **eval/at** seems like a step backward from **eval/ce**, we can still (almost) simulate one with the other by doing this:

```
(define (eval/at x) (eval/ce x))
```

Unfortunately, there is one variable too many in the environment that's been captured: the local variable named **x** hides the global variable of the same name. [see Ex. 8.3] This problem might make you think that a more refined way of handling the environment could solve the difficulty, so this definition suggests to us how we should define **eval/at** in the interpreter that compiles byte-code. The new definition again uses the function **compile-and-run** but without asking it to push the return address because the caller of **eval/at** has already done it. The compilation, however, is done here with **r.init** since we no longer have access to the current lexical environment.

```
(definitial eval
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
```

```
(make-primitive
 (lambda ()
   (if (= arity+1 (activation-frame-argument-length *val*))
       (let ((v (activation-frame-argument *val* 0)))
         (if (program? v)
             (compile-and-run v r.init #t)
             (signal-exception #t (list "Illegal program" v)) ) )
       (signal-exception #t (list "Incorrect arity" 'eval)) ) ) ) )
```

Evaluating as if we were at the toplevel loop should not be taken too literally because of the dynamic environment. For example, whether **eval** is a special form or a function, we should still observe this:

```
(dynamic-let (a 22)
  (eval '(dynamic a)) )   → 22
```

As far as the initial contract of **eval** with respect to equation (1), it begins in the empty context, that is, right at the fundamental level. In contrast, **eval/at** does not satisfy equation (2), as those earlier examples prove. To be more specific about the behavior of **eval/at**, we'll turn to equation (3) where v is a variable that cannot be captured:

$$C[\texttt{(eval/at '}\pi\texttt{)}] \equiv \texttt{(let ((}v\texttt{ (lambda () }\pi\texttt{))) }C[(v)]) \qquad (3)$$

The variation of **eval** that you'll encounter most commonly among Lisp systems is **eval/at**.

8.5 The Cost of eval

The overall cost of using **eval** is hard to grasp. Suppose first of all that the autonomous application that we are about to construct is none other than the famous **(display "Hello world")**. If the special form **eval/ce** does not occur in the application (and that's something we can simply check statically) then there's no cost! If it occurs, then we must at least add the compiler to the application, so its size changes by an order of magnitude; let's say roughly 50 to 500 kilobytes. If we get dynamic evaluation in the form of a function, and if it is not proved that the function will not be useless (in short, if we need the function) then we're back to the same case. To prove that **eval/at** is not needed, we must verify that it is never used. The language is helpful for this proof: in Scheme, for example, it suffices to prove that the global variable **eval/at** is not mentioned. In COMMON LISP, that's generally not possible since we must prove in addition (and among other things) that nobody has generated the character string **"eval/at"**, converted it to a symbol by means of **find-symbol**, and (by means of **symbol-function**) extracted the function of the same name from that symbol. It's costly to detect even just the mention of **symbol-function** with an argument that we cannot foresee.

If by chance a call to **eval** occurs, can we limit the set of values to which **eval** is applied? In most cases, this analysis is hardly possible, and you might even think that any value is possible. Consequently, we can no longer remove a single function from the global environment since *anything* and *everything* might be called. The generation of an application thus must contain everything that the

language defines, with no exceptions, and the size of the executable, especially in a language as rich as COMMON LISP, grows another order of magnitude.

We're not past the worst yet. Even if we are in the ideal case with a system of modules that limit where the global environment can be seen and whether it can be modified, as in [QP91b], the very presence of **eval** kills off the possibility of many ways of improving compilation. Look at the following definition:

```
(define (fib n)
  (if (< n 2) (eval (read))
      (+ (fib (- n 1)) (fib (- n 2))) ) )
```

Independently of the effects that **eval** might have on other global variables, **eval** can also modify the global variable **fib**. That fact obliges us, during the second recursive call, to search for the value of **fib** by the address associated with the global variable, rather than using a blind **GOTO**. Since **eval** can modify anything, we don't even have the usual advantages of global variables being invariable. In short, the presence of even one single **eval** in a module makes it possible to modify practically anything in that module.

For all those reasons, **eval** is considered an expensive characteristic. However, inside a large application (something like a few megabytes) and in development where everything is in a more or less unstable state, **eval** is a low-level, useful addition. True: if we have only a minor little calculation to program, it is simpler (in terms of how long it takes to program and how costly a programmer's time is) to call **eval** to compute arithmetic expressions, rather than to implement an interpreter in an *ad hoc* language. Lisp is a remarkable extension language, and making **eval** available in a library is a major advantage.

8.6 Interpreted eval

This book uses an immoderate number of interpreters. In case there were no **eval** in a system, you might ask whether a user couldn't supply one by his or her own means. After all, given the number of interpreters you've already seen, you would only have to choose one from the many available, pick up the code by FTP (or just type it in yourself), and put it into your application. Well, yes and no.

In pure Scheme, if you want to write an evaluator, it can only be a function since you can't define your own special forms there. Two problems then arise.

8.6.1 Can Representations Be Interchanged?

The first problem we alluded to is how to organize interactions between the underlying system and the interpreter that you want to write. This problem means that you can't impose new data types and that in fact you must conform to existing types. The pairs that the interpreter manages must be the same pairs as the implementation; Booleans must be the same; functions have to respect a similar calling protocol. Explicit evaluation, which appears in the following example, must necessarily return a function that can be invoked by the underlying system.

```
((eval '(lambda (x) (cons x x)))
 33 )     → (33 . 33)
```

Conversely, the interpreter must also know how to invoke functions from the underlying system, like this:

```
(((eval '(lambda (f)
           (lambda (x) (f x)) ))
  list )
 44 )     → (44)
```

One possible common representation for functions is to use functions with multiple arity (`lambda values ...`) in a way that adapts to the two possible invocation modes. The modifications that this entails for the first interpreter in this book (the naive interpreter of the first chapter [see p. 3]) for example, are simple. You can easily deduce the others from this:

```
(define (invoke fn args)
  (if (procedure? fn)
      (apply fn args)
      (wrong "Not a function" fn) ) )
(define (make-function variables body env)
  (lambda values
     (eprogn body (extend env variables values)) ) )
```

A subtle point is to make sure that error handling (for example, errors about arity) is the same in `eval` and in the underlying system, but we'll skip that detail.

8.6.2 Global Environment

The second problem we alluded to concerns how to handle the global environment. The evaluators in this book all have their own definition of their global environment; they build it by means of such macros as `definitial`, `defprimitive`, and others. Here, the problem is to cooperate with the underlying system to insure that the global environment whether seen from the interpreter or from the system is the same. In other words, the following expression has to be evaluated without error:

```
(begin (set! foo 128)
       (eval '(set! bar (+ foo foo)))
       bar )    → 256
```

Compiler for Autonomous Applications

In a compiler that works on files, like Scheme→C in [Bar89] or Bigloo in [Ser94], the program is known in advance and by construction it does not know how to access variables that it does not mention. Thus the variables that `eval` creates can be handled only by `eval`. Consequently, all we have to do is connect the underlying global environment to the interpreter, and to do so, we define two functions known as `global-value` and `set-global-value!`. Their body is systematically formed after all the global variables of the program, and they are known statically. We can even imagine that these functions[6] are synthesized automatically. [see Ex. 7.3]

```
(define (global-value name)
```

6. In the function `set-global-value!`, the variable `car` does not occur in order to preserve its immutability.

```
    (case name
      ((car) car)
      ...
      ((foo) foo)
      (else (wrong "No such global variable" name)) ) )
(define (set-global-value! name value)
  (case name
    ((foo) (set! foo value))
    ...
    (else (wrong "No such mutable global variable" name)) ) )
```

The interpreter can create as many global variables as it wants; none of them can be reached directly by the underlying system; only the interpreter can manipulate them. This is not a problem since the underlying system does not mention them so it may safely continue to ignore them.

Interactive System

In contrast, if we're in a system that has an interactive loop, then the system and the interpreter can both create new variables. The example we gave earlier can explode into a sequence of interactions where we see that we can ask the underlying system the value of a variable created by the interpreter and vice-versa, like this:

```
? (begin (set! foo 128)
         (eval '(set! bar (+ foo foo)))
         2001 )
= 2001
? bar
= 256
```

There are several solutions to this problem of cooperation.

Using Symbols

The most conventional of these solutions uses symbols, a practice that has for years undermined the semantic basis of Lisp because it regrettably confuses symbols with variables, a confusion made worse by the fact that variables are represented by symbols and that (as you will see) symbols can implement variables.

Symbols are data structures that need a unique field to associate the symbol with its name, a character string. Symbols are created explicitly in Scheme by the function **`string->symbol`**. Two symbols of the same name cannot exist simultaneously and still be different. We usually insure that point with a hash table associating character strings with their symbol. The function **`string->symbol`** begins by searching in the hash table to see whether a symbol by that name already exists and if so, the function returns it; otherwise, the function builds the symbol. To make searching for symbols by name faster, supplementary (hidden) fields are often added to connect symbols to each other.

We often add a property list to symbols. It is usually managed as a P-list, but you'll also encounter A-lists and hash tables in this role as well. How large they are and how fast you can access them depends on the implementation of property lists. Some Lisp systems, such as Le-Lisp have even added fields to symbols to

serve as caches for properties that are widely used, such as, for example, how to pretty-print forms beginning with that symbol.

Since the structure of a symbol is nothing more than a few fields, why don't we add another field there to store the global value of the variable of the same name? In the case of Lisp_2, why don't we store the value of the global function of the same name as well? In fact, since time immemorial, that is exactly what is done under the names of Cval and Fval, as in [Cha80, GP88]. Functions exist to read and write these fields. In COMMON LISP, they are **symbol-value** and **(setf symbol-value)** as well as **symbol-function** and **(setf symbol-function)**.

This technique is particularly attractive because it is exceedingly sure. Every symbol that might serve as the representation of a global variable necessarily has been built beforehand by **symbol->string** (generally by **read**) so an address to contain its global value exists already as a field in the symbol. Of course, this arrangement is wasteful since not every symbol necessarily supports the representation of a global variable of the same name; on the other hand, no global variable exists without being associated with an address to contain its global value.

Thus it suffices to provide two functions, **global-value** and **set-global-value!**, to get and assign the value of global variables, starting from their names. The explicit interpreter will handle the global environment through these functions, and the problem of extending this environment never comes up since it is resolved by the magical underlying mechanism connected to the idea of a symbol.

Let's indicate how to enrich the interpreter from the first chapter to take into account these magic functions. The environment, the value of the variable **env**, will contain only local variables to the exclusion of global variables which will be managed directly. Since we have adopted a common representation of functions between the interpreter and the system, we can drop the definition of the global environment entirely from the interpreter because the interpreter can now use the global environment of the system directly. Thus we have this:

```
(define (lookup id env)
  (if (pair? env)
      (if (eq? (caar env) id)
          (cdar env)
          (lookup id (cdr env)) )
      (global-value id) ) )
(define (update! id env value)
  (if (pair? env)
      (if (eq? (caar env) id)
          (begin (set-cdr! (car env) value)
                 value )
          (update! id (cdr env) value) )
      (set-global-value! id value) ) )
```

This solution seems elegant, but at an impressive cost since every global variable is thus accessible by name. An autonomous application containing a call to **global-value** cannot eliminate anything from the library of initial functions since *a priori* every name can be computed and thus everything is necessary. That's not too annoying in a programming environment since one of its properties is to make everything available to the programmer. However, it is a real problem for a

small autonomous application. Even worse, the function `set-global-value!` can change the value of any global variable and thus break any optimization of the compilation since nothing is any longer *a priori* immutable.

The two functions, `global-value` and `set-global-value!`, can be considered as one specialized version of `eval`. By the way, we found it in old Lisp systems under the name `symeval`. You can also see those two functions as reflective functions that reify access to the global environment. Writing "`foo`" in order to know the value of the global variable `foo` is the same as writing and explicitly evaluating `(global-value 'foo)`.

First Class Environment

Another technique is to define an `eval` function with two variables: the expression to compute and the environment in which to compute it. The binary `eval` of the naive evaluator in the first chapter has a signature like that, but the shortcoming in adopting that solution is that we must be able to provide values as the second argument and environments too, although no operation is available for getting them. Consequently, we must allow reification of environments as well as possibly other operations, such as extraction or modification of the value of variables. By means of these operations, we can connect the environments handled by the explicit evaluator with the environments of the implementation. This technique raises many problems that we'll address now.

8.7 Reifying Environments

The denotational interpreter clearly demonstrated that in Scheme, evaluation depends on a triple: environment, continuation, memory. Once again rejecting memory, we see that continuations can be handled via `call/cc` or `bind-exit` (which reifies them). That was not the case for lexical environments, which could not be accessed directly. The next section covers how to reify them in data structures that can be manipulated.

The implementations of first class environments are characterized and differentiated, according to [RA82, MR91], by their properties and their operations. The three fundamental operations of a first class environment are: searching for the value of a variable; modifying the value of a variable extending the environment. To reify the environment is to provide a means of capturing bindings but without the abstractions that capture them only in an opaque way or in a way that cannot be manipulated.

8.7.1 Special Form `export`

We'll introduce a special form, named `export`; we'll mention to it the names of variables to capture, and it will return an environment enclosing their bindings with the specified variables. The reified environment can be used as the second argument of the binary evaluation function; we'll distinguish that function from earlier ones by naming it `eval/b`. Let's take a few examples.

```
(let ((r (let ((x 3) (export x)))))
```

```
  (eval/b 'x r) )                          → 3
(let ((f+r (let ((x 3))
             (cons (lambda () x) (export x)) )))
  (eval/b '(set! x (+ 1 x)) (cdr f+r))
  ((car f+r)) )                            → 4
(let ((r (export car cons)))
  (let ((car cdr))
    (eval/b '(car (cons 1 2)) r) ) )   → 1
```

In the first example, the first class environment that's being created captures the variable **x** which can thus be used at leisure by **eval/b**. The second example shows that we can also modify the variable **x**, and that it really is the binding being captured since the modification is perceived from what encloses it by the normal means of abstraction. The third example shows that we can also capture global bindings.

The special form **export** lets us juggle environments, so to speak, by capturing the purposes of environments and using them elsewhere. Its implementation is not trivial, so we'll describe it. As we do for all special forms, we'll add a clause for it to **meaning**, the function that analyzes syntax, like this:

```
... ((export) (meaning-export (cdr e) r tail?)) ...
```

A reified environment must not only capture the list of activation records but also save the names and addresses of variables that are supposed to be captured. That aspect recalls the implementation of **eval/ce** which required us to save the same information. We'll represent a reified environment by an object with two fields: the list of activation records and a list of name-address pairs that serve as the "roadmap" to the activation records. You can see what we mean in Figure 8.2.

```
(define-class reified-environment Object
  ( sr r ) )
```

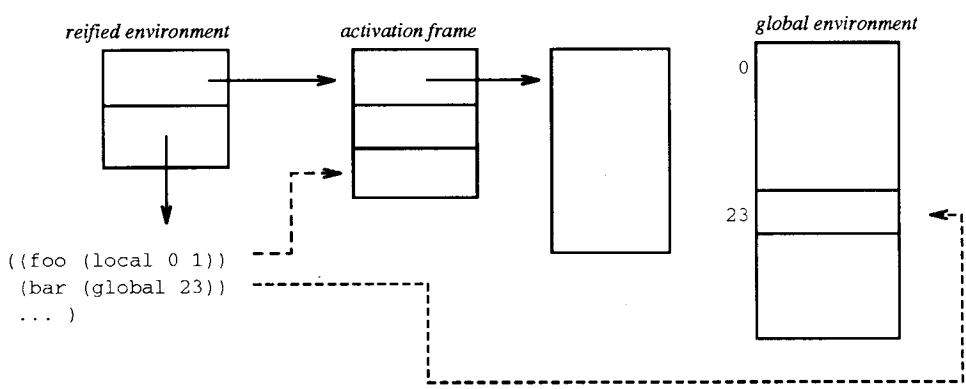


Figure 8.2 Reified environment

Without going into the details right away, we will compile an **export** form like this:

```
(define (meaning-export n* r tail?)
  (unless (every? symbol? n*)
          (static-wrong "Incorrect variables" n*) )
  (append (CONSTANT (extract-addresses n* r)) (CREATE-1ST-CLASS-ENV)) )
(define (CREATE-1ST-CLASS-ENV) (list 254))

(define-instruction (CREATE-1ST-CLASS-ENV) 254
  (create-first-class-environment *val* *env*) )

(define (create-first-class-environment r sr)
  (set! *val* (make-reified-environment sr r)) )
```

The list of name-address pairs will be represented by a quotation; the new instruction **CREATE-1ST-CLASS-ENV** will take that quotation in the register ***val*** to build the object that we want.

To simplify the evaluation, we will change the representation of static environments for compilation (the values of variables **r**). In place of the representation as a rib cage, we'll adopt a more explicit representation as an association list of names-addresses like the one in Figure 8.2. There is little to change, but by simplifying **compute-kind**, we complicate **r-extend***: every time we extend the environment, it must bury the deep variables a little deeper.

```
(define (compute-kind r n)
  (or (let ((var (assq n r)))
        (and (pair? var) (cadr var)) )
      (global-variable? g.current n)
      (global-variable? g.init n)
      (adjoin-global-variable! n) ) )
(define r.init '())
(define (r-extend* r n*)
  (let ((old-r (bury-r r 1)))
    (let scan ((n* n*)(i 0))
      (cond ((pair? n*) (cons (list (car n*) `(local 0 . ,i))
                              (scan (cdr n*) (+ i 1)) ))
            ((null? n*) old-r)
            (else (cons (list n* `(local 0 . ,i)) old-r)) ) ) ) )
(define (bury-r r offset)
  (map (lambda (d)
         (let ((name (car d))
               (type (car (cadr d))) )
           (case type
             ((local checked-local)
              (let* ((addr (cadr d))
                     (i (cadr addr))
                     (j (cddr addr)) )
                `(,name (,type ,(+ i offset) . ,j) . ,(cddr d)) ) )
             (else d) ) ) )
       r ) )
```

Rather than reify an environment with only those variables that are mentioned, we will make **(export)** equivalent to the form **(export** *variables* ... **)** where we will already have specified all the variables of the ambient abstractions. With the

form `(export)`, which is often called `(the-environment)`, `eval/b` can be simulated entirely by `eval/ce` since:

$$\texttt{(eval/ce '}\pi\texttt{)} \quad \equiv \quad \texttt{(eval/b '}\pi\texttt{ (export))}$$

To achieve that improvement, all we have to do is capture every available environment (which now has the right structure) and write this:

```
(define (extract-addresses n* r)
  (if (null? n*) r
      (let scan ((n* n*))
        (if (pair? n*)
            (cons (list (car n*) (compute-kind r (car n*)))
                  (scan (cdr n*)) )
            '() ) ) ) )
```

8.7.2 The Function `eval/b`

There are still traces of `eval/ce` inside `eval/b`. Those two are similar as far as how they get evaluation parameters and with respect to the return protocol for evaluation. Indeed, it's useful to compare what follows with what went before. The function `eval/b` verifies the nature of its arguments, then delegates the function `compile-on-the-fly` (which you've already seen) to take care of first compiling the expression in the environment that's provided, then installing the code somewhere in memory, and executing it. The current environment doesn't need to be saved since that save has already been taken care of by the calling protocol for functions.

```
(definitial eval/b
  (let* ((arity 2)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let ((exp (activation-frame-argument *val* 0))
                 (env (activation-frame-argument *val* 1)) )
             (if (program? exp)
                 (if (reified-environment? env)
                     (compile-and-evaluate exp env)
                     (signal-exception
                      #t (list "Not an environment" env) ) )
                 (signal-exception #t (list "Illegal program" exp)) ) )
           (signal-exception
            #t (list "Incorrect arity" 'eval/b) ) ) ) ) ) )
(define (compile-and-evaluate v env)
  (let ((r (reified-environment-r env))
        (sr (reified-environment-sr env)) )
    (set! *env* sr)
    (set! *pc* (compile-on-the-fly v r)) ) )
```

8.7.3 Enriching Environments

Since environments are extended so often, it makes sense to get this operation for ourselves, too. We will thus furnish a function, **enrich**, which takes an environment and the names of variables to add to it. The return value will be a *new* enriched environment. In fact, **enrich** is a functional modifier that does not disturb its arguments. Let's look at an example of how it is used. We'll simulate **letrec** by explicitly enriching the environment. A global binding[7] will be captured; two local variables, **odd?** and **even?** will enrich it; two mutually recursive definitions will be evaluated there; a computation will be carried out eventually.

```
((lambda (e)
   (set! e (enrich (export *) 'even? 'odd?))
   (eval/b '(set! even? (lambda (n) (if (= n 0) #t (odd? (- n 1))))) e)
   (eval/b '(set! odd? (lambda (n) (if (= n 0) #f (even? (- n 1))))) e)
   (eval/b '(even? 4) e) )
 'ee )                                  → #t
```

Figure 8.3 shows the results of this construction in detail. A new activation record is allocated to contain the new variables. The reified environment associates these new names with the appropriate addresses. The only difficulty is that these new variables have no values, even though the variables exist. Here we find ourselves back in the discussion about **letrec** or about non-initialized variables. **[see p. 60]**

We'll thus introduce a new type of local address, **checked-local**, which is almost analogous to **checked-global**. Our definition of **enrich** brings us the possibility of non-initialized local variables—a situation that was not possible with **lambda**. One fix would be to force environments to be enriched by variables accompanied by their values.

Programming **enrich** is simple now, if not short:

```
(definitial enrich
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (>= (activation-frame-argument-length *val*) arity+1)
           (let ((env (activation-frame-argument *val* 0)))
             (listify! *val* 1)
             (if (reified-environment? env)
                 (let* ((names (activation-frame-argument *val* 1))
                        (len (- (activation-frame-argument-length *val*)
                                2 ))
                        (r (reified-environment-r env))
                        (sr (reified-environment-sr env))
                        (frame (allocate-activation-frame
                                (length names) )) )
                   (set-activation-frame-next! frame sr)
                   (do ((i (- len 1) (- i 1)))
                       ((< i 0))
```

7. Ahem, there is a design error here: there is no means to create an empty environment with **export**, so we capture just any old thing—in this case, multiplication—to create a little environment, as in [QD96].

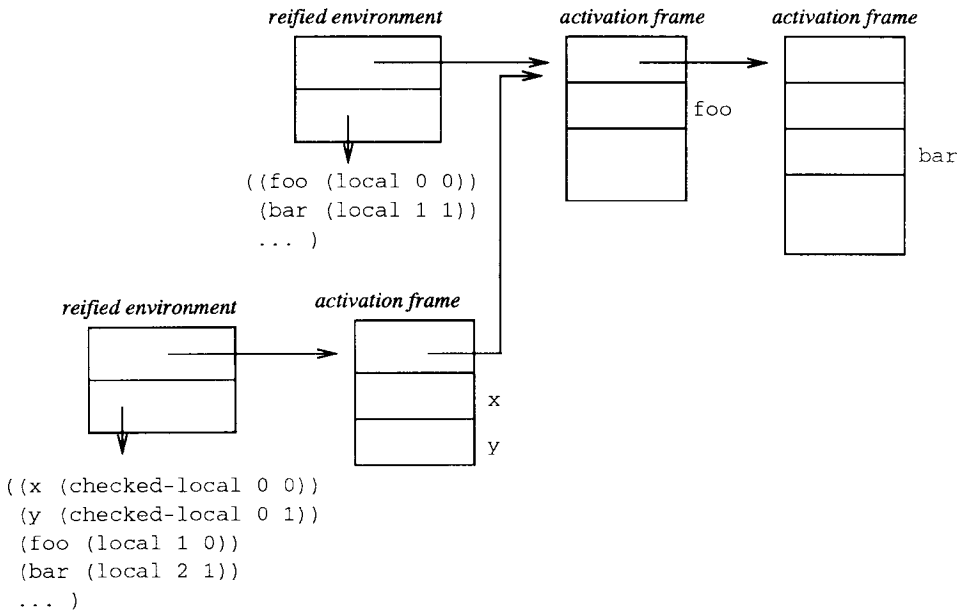


Figure 8.3 Enriched environment (`enrich` *env* `'x 'y`)

```
                    (set-activation-frame-argument! frame i
                                                    undefined-value ) )
                  (unless (every? symbol? names)
                    (signal-exception
                     #f (list "Incorrect variable names" names ) ) )
                  (set! *val* (make-reified-environment
                               frame
                               (checked-r-extend* r names) ))
                  (set! *pc* (stack-pop)) )
                 (signal-exception
                  #t (list "Not an environment" env) ) ) )
           (signal-exception
            #t (list "Incorrect arity" 'enrich) ) ) ) ) ) )
(define (checked-r-extend* r n*)
  (let ((old-r (bury-r r 1)))
    (let scan ((n* n*)(i 0))
      (cond ((pair? n*) (cons (list (car n*) `(checked-local 0 . ,i))
                              (scan (cdr n*) (+ i 1)) ))
            ((null? n*) old-r) ) ) ) )
```

However, we must upate the two generators, `meaning-reference` and `meaning-assignment`, to take into account this new type of address (`checked-local`).

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
    (if kind
        (case (car kind)
```

```
            ((checked-local)
             (let ((i (cadr kind))
                   (j (cddr kind)) )
               (CHECKED-DEEP-REF i j) ) )
            ((local)
             (let ((i (cadr kind))
                   (j (cddr kind)) )
               (if (= i 0)
                   (SHALLOW-ARGUMENT-REF j)
                   (DEEP-ARGUMENT-REF i j) ) ) )
            ((global)
             (let ((i (cdr kind)))
               (CHECKED-GLOBAL-REF i) ) )
            ((predefined)
             (let ((i (cdr kind)))
               (PREDEFINED i) ) ) )
          (static-wrong "No such variable" n) ) ) )
(define (meaning-assignment n e r tail?)
  (let ((m (meaning e r #f))
        (kind (compute-kind r n)) )
    (if kind
        (case (car kind)
          ((local checked-local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-SET! j m)
                 (DEEP-ARGUMENT-SET! i j m) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (GLOBAL-SET! i m) ) )
          ((predefined)
           (static-wrong "Immutable predefined variable" n) ) )
        (static-wrong "No such variable" n) ) ) )
```

We must not forget to add the instruction `CHECKED-DEEP-REF` to our byte-code machine, like this:

```
(define-instruction (CHECKED-DEEP-REF i j) 253
  (set! *val* (deep-fetch *env* i j))
  (when (eq? *val* undefined-value)
    (signal-exception #t (list "Uninitialized local variable")) ) )
```

8.7.4 Reifying a Closed Environment

Some interpreters make primitives available to extract an environment from a closure that contains it. Next to the function `procedure->environment`, these interpreters often add the function `procedure->definition`, which extracts the definition from a closure. These two functions are simple to implement in an interpreter, but in a compiler, they become more complicated and consume more memory. In effect, we have to do two things at once: one, store definitions (which adds to quota-

tions) and two, reify closed environments (which necessitates saving their structure and adding new quotations). Moreover, the function **procedure->environment** is very indiscrete since we might modify a closed environment when we reify it—a change that prohibits any optimization of local variables because under these conditions even local things can be reached. In that sense, **procedure->environment** is much more costly than **export** because it strips the evaluator without putting any limitations on what can be done with it, whereas **export** restricts what can be (un)done to only those variables that are mentioned.

The function **procedure->definition** is useful for introspection. With it, we can provide a debugger that can dissect functions to apply. Here is an example of how to use those two functions. The function **trace-procedure1** takes a unary function, examines it, and rebuilds a new unary function, comparable to the first one but printing its argument on input and its result on output.

```
(define (trace-procedure1 f)
  (let* ((env (procedure->environment f))
         (definition (procedure->definition f))
         (variable (car (cadr definition)))
         (body (cddr definition)) )
    (lambda (value)
      (display (list 'entering f 'with value))
      (eval/b `(begin (set! ,variable ',value)
                      (let ((result (begin . ,body)))
                        (display (list 'result 'is result))
                        result ) )
              (enrich env variable) ) ) ) )
```

In fact, that program cheats a little. A unary function like **car** will upset **trace-procedure1**. The synthesized program for **eval/b** is not even a legal program because it quotes the value of **value**, which is not always a value that can be legitimately quoted. However, the function **trace-procedure1** produces an important effect for debugging. That effect results from combining introspection functions for closures, first class environments, and explicit evaluation.

If we suggest a way of implementing these functions, you can judge their cost for yourself. The function **procedure->definition** is the simpler: all it has to do is associate any closure with the expression that defines that closure, that is, a quotation. The problem: where to put that quotation since there may be many closures associated with the same definition. The salient characteristic of a closure is the address of its code. All the closures of a given abstraction share that same address, so we'll associate that address with the appropriate quotation. To do so, we'll use a technique well known to programmers in asssembly language: we'll insert data in the instructions. Since our machine forbids that, we'll actually insert quotations as in Figure 8.4.

Programming it is trivial. Once again we must change the code generators since now we must provide the definition and the static compilation environment to them because the functions **procedure->environment** and **procedure->definition** need that information. The version for an n-ary function is easy[8] to deduce:

8. Using **EXPLICIT-CONSTANT** means we do not get a **(PREDEFINEDO)** in place of a **(CONSTANT '())**.

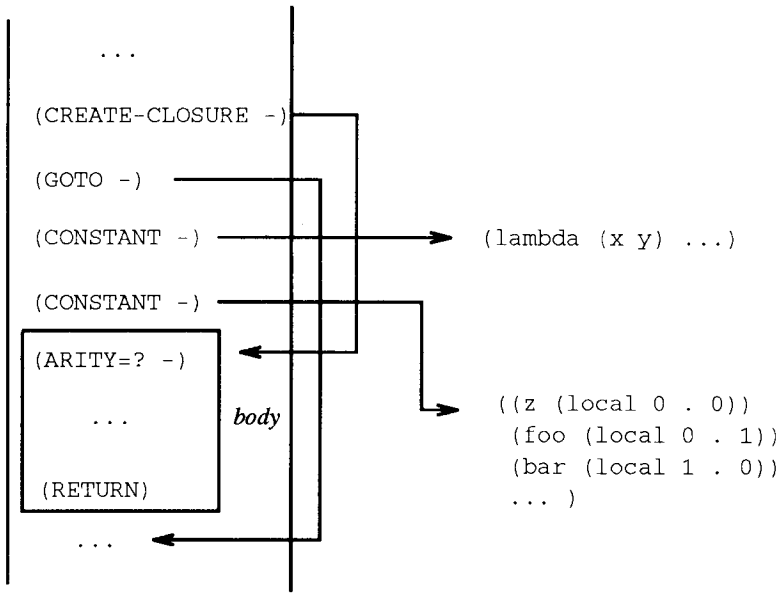


Figure 8.4 Abstraction that can be dissected

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let* ((arity (length n*))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence e+ r2 #t)) )
    (REFLECTIVE-FIX-CLOSURE m+ arity '(lambda ,n* . ,e+) r) ) )
(define (REFLECTIVE-FIX-CLOSURE m+ arity definition r)
  (let* ((the-function (append (ARITY=? (+ arity 1)) (EXTEND-ENV)
                               m+  (RETURN) ))
         (the-env (append (EXPLICIT-CONSTANT definition)
                          (EXPLICIT-CONSTANT r) ))
         (the-goto (GOTO (+ (length the-env) (length the-function)))) )
    (append (CREATE-CLOSURE (+ (length the-goto) (length the-env)))
            the-goto the-env the-function ) ) )
```

The function **procedure->definition** finds the definition of a closure by the quotation located two instructions before the code of its body.

```
(definitial procedure->definition
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (>= (activation-frame-argument-length *val*) arity+1)
           (let ((proc (activation-frame-argument *val* 0)))
             (if (closure? proc)
                 (let ((pc (closure-code proc)))
                   (set! *val* (vector-ref
```

```
                        *constants*
                        (vector-ref *code* (- pc 3)) ))
                  (set! *pc* (stack-pop)) )
                (signal-exception #f (list "Not a procedure" proc)) ) )
          (signal-exception
           #t (list "Incorrect arity" 'enrich) ) ) ) ) ) )
```

The function **procedure->environment** extracts the closed environment from the closure and reifies it by adding the name-address mapping to it, making it intelligible. This mapping is found one instruction before the code for its body.

```
(definitial procedure->environment
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (>= (activation-frame-argument-length *val*) arity+1)
           (let ((proc (activation-frame-argument *val* 0)))
             (if (closure? proc)
                 (let* ((pc (closure-code proc))
                        (r (vector-ref
                            *constants*
                            (vector-ref *code* (- pc 1)) )) )
                   (set! *val* (make-reified-environment
                                (closure-closed-environment proc)
                                r ))
                   (set! *pc* (stack-pop)) )
                 (signal-exception #f (list "Not a procedure" proc)) ) )
           (signal-exception
            #t (list "Incorrect arity" 'enrich) ) ) ) ) ) )
```

In short, adding the functions **procedure->definition** and **procedure->environment** makes it possible for programs to understand their own code. They also make it possible to write introspective debugging tools. However, they both have a non-negligible cost because of the amount of information to save. Above all, they are both totally indiscrete since if all things can be reified, then nothing can be definitively hidden. You can no longer sell code that you want to keep secret, nor can you hide the implementation of certain types of data, nor even hope that certain local optimizations can take place since nothing can any longer be guaranteed constant.

Nevertheless, we can offset some of these faults by creating a new special form, say, **reflective-lambda**, with the following syntax:

(reflective-lambda (*variables*) (*exportations*)
body **)**

Like the abstractions that they generalize, their variables are specified in the first parameter and their body in the last ones. Between those two, the exportation clause sets the only free variables in the body which will be available for inspection from the outside in the environment that this abstraction encloses. Accordingly, a normal abstraction **(lambda** (*variables*) *body*) is none other than **(reflective-lambda** (*variables*) () *body*); that is, it doesn't export anything.

By making only a few bindings visible, we can conceal others and thus protect ourselves from possible indiscretions.

There is yet another problem with the inquisitive function **procedure->environment**. Exactly which environment does the abstraction in the following example enclose anyway?

```
(let ((x 1)(y 2))
  (lambda (z) x) )
```

The enclosed environment certainly contains **x**, which is free in the body of the abstraction. Does it capture **y**, which is present in the surrounding lexical environment? The way of programming that we looked at earlier also captures **y** because we can assume that the contract from **procedure->environment** lets us evaluate any expression (even one containing **x** and **y**) as if it were found in the lexical environment where the closure was created. It's not so much the closed environment that **procedure->environment** returns as the entire lexical environment from the creation of the closure.

8.7.5 Special Form `import`

Quite often the form on which to call the evaluator has a static structure. That was the case, for example, in the procedure **trace-procedure1**. Rather than recompile on the fly, we might imagine a kind of precompilation that makes it possible to branch on an environment which would not be available until execution. The new special form **import** meets that contract. It looks like this:

(import (*variables*) *environment forms* **)**

The special form **import** evaluates the *forms* that make up its body in a rather special lexical environment. The names of free variables of the body appearing in the list[9] *variables* are the ones to take in the *environment*; the others are to take in the lexical environment of the call to **import**. The list of variables is static, not computed; *environment* is evaluated first, then the *forms*.

Here is an example, inspired by MEROONET where we had the problem of representing generic functions which are simultaneously objects and functions. If we allow access to enclosed variables, then a closure can be handled like an object where the fields are its free variables. This idea is the basis[10] for identifying closures with objects. In the implementation of MEROONET, to add a method to a generic function, we write this method in the right place in the vector of methods. Assuming that a generic function encloses this vector under the name **methods**, we can write it directly like this:

```
(define (add-method! generic class method)
  (import (methods) (procedure->environment generic)
    (vector-set! methods (Class-number class) method) ) )
```

In that example, you can see the usefulness of the special form **import** with respect to the function **eval/b**. The references to the variables **class** and **method** are static while only the variable **methods** still floats; it can be resolved only when the second parameter of **import** is known. Thus we've gained compilation on the

9. No variables could mean that all of them are captured as in [QD96].
10. Other facilities, like specializing the class of functions, are still missing.

fly like **eval/b** executed, and we see that we no longer synthesize programs to compile: **import** is a kind of **eval** with its fat removed; it's more static since we know statically the only references that still have to be resolved. In that sense, **import** belongs to the kind of quasi-static binding proposed in [LF93] or in [NQ89]. Its very name suggests its close relation to its "cousin" **export**: one produces the environments that the other branches to. We're getting nearer here to the rudiments of first class modules.

How should we implement **import** as a special form? First, we'll add a line to the syntactic analyzer, **meaning**, to recognize this new special form.

```
... ((import) (meaning-import (cadr e) (caddr e) (cdddr e) r tail?)) ...
```

The associated byte-code generator stores the list of floating variables on top of the stack, evaluates the environment which will end up in the register ***val***, executes the new instruction **CREATE-PSEUDO-ENV**, and procedes to the evaluation of the body of the **import** form, whether in tail position or not.

```
(define (meaning-import n* e e+ r tail?)
  (let* ((m (meaning e r #f))
         (r2 (shadow-extend* r n*))
         (m+ (meaning-sequence e+ r2 #f)) )
    (append (CONSTANT n*) (PUSH-VALUE) m (CREATE-PSEUDO-ENV)
            (if tail? m+ (append m+ (UNLINK-ENV))) ) ) )
```

The body of the special form **import** is evaluated by constructing of a pseudo-activation record (an instance of the class **pseudo-activation-frame**); its size is the number of floating variables mentioned in the form. It behaves like an environment in the sense that we can connect them just like activation records, but in fact it contains real addresses where to look for the floating variables as well as the environment from which to extract them. During the creation of this pseudo-activation-frame, we use **compute-kind** to compute where the floating variables are, and we put its address (not its value) into the pseudo-frame.

```
(define-instruction (CREATE-PSEUDO-ENV) 252
  (create-pseudo-environment (stack-pop) *val* *env*) )

(define-class pseudo-activation-frame environment
  ( sr (* address) ) )
(define (create-pseudo-environment n* env sr)
  (unless (reified-environment? env)
    (signal-exception #f (list "not an environment" env)) )
  (let* ((len (length n*))
         (frame (allocate-pseudo-activation-frame len)) )
    (let setup ((n* n*)(i 0))
      (when (pair? n*)
        (set-pseudo-activation-frame-address!
         frame i (compute-kind (reified-environment-r env) (car n*)) )
        (setup (cdr n*) (+ i 1)) ) )
    (set-pseudo-activation-frame-sr! frame (reified-environment-sr env))
    (set-pseudo-activation-frame-next! frame sr)
    (set! *env* frame) ) )
```

The body of the special form **import** has to be specially compiled with respect to floating variables. Those variables are scattered through the compilation envi-

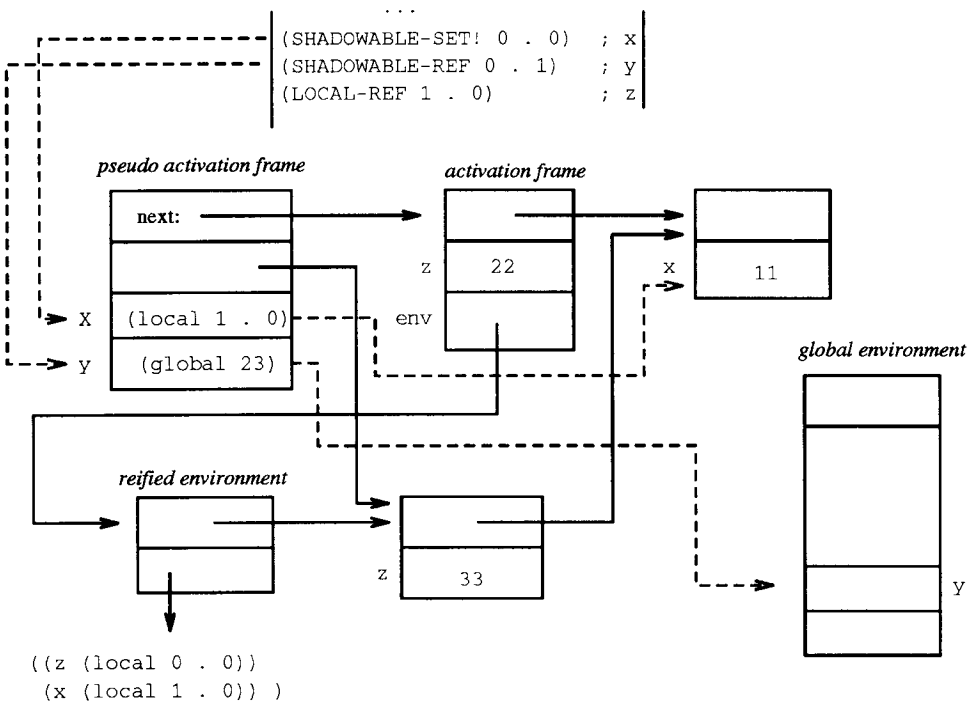


Figure 8.5 Environments in `import`

ronment `r` by the special extension function, `shadow-extend*`. As the old saying goes, there's no computing problem that a well placed indirection can't solve: the address of the floating variables will be found by means of one indirection. We have to search for the value of a floating variable, we will find its address in the pseudo-activation record, and we'll use that address to extract the value from the environment provided explicitly for that. You can see those ideas in Figure 8.5 for the following program:

```
(let ((x 11))
  (let ((z 22)
        (env (let ((z 33)) (export))) )
    (import (x y) env
      (list (set! x y) z) ) ) )
```

Floating variables will be compiled in a special way, so they are specially marked by `shadow-extend*` in the compilation environment.

```
(define (shadow-extend* r n*)
  (let enum ((n* n*)(j 0))
    (if (pair? n*)
        (cons (list (car n*) '(shadowable 0 . ,j))
              (enum (cdr n*) (+ j 1)) )
        (bury-r r 1) ) ) )
```

We'll modify **meaning-reference** as well as **meaning-assignment** to accept these new types of variables, and we'll invent two new instructions, **SHADOW-REF** and **SHADOW-SET!**, to manage access to them.

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
    (if kind
        (case (car kind)
          ((checked-local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (CHECKED-DEEP-REF i j) ) )
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-REF j)
                 (DEEP-ARGUMENT-REF i j) ) ) )
          ((shadowable)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (SHADOWABLE-REF i j) ) )
          ((global)
           (let ((i (cdr kind)))
             (CHECKED-GLOBAL-REF i) ) )
          ((predefined)
           (let ((i (cdr kind)))
             (PREDEFINED i) ) ) )
        (static-wrong "No such variable" n) ) ) )

(define-instruction (SHADOW-REF i j) 231
  (shadowable-fetch *env* i j) )

(define (shadowable-fetch sr i j)
  (if (= i 0)
      (let ((kind (pseudo-activation-frame-address sr j))
            (sr (pseudo-activation-frame-sr sr)) )
        (variable-value-lookup kind sr) )
      (shadowable-fetch (environment-next sr) (- i 1) j) ) )

(define (variable-value-lookup kind sr)
  (if (pair? kind)
      (case (car kind)
        ((checked-local)
         (let ((i (cadr kind))
               (j (cddr kind)) )
           (set! *val* (deep-fetch sr i j))
           (when (eq? *val* undefined-value)
             (signal-exception
              #t (list "Uninitialized local variable") ) ) ) )
        ((local)
         (let ((i (cadr kind))
               (j (cddr kind)) )
```

```
      (set! *val* (if (= i 0)
                      (activation-frame-argument sr j)
                      (deep-fetch sr i j) )) ) )
    ((shadowable)
     (let ((i (cadr kind))
           (j (cddr kind)) )
       (shadowable-fetch sr i j) ) )
    ((global)
     (let ((i (cdr kind)))
       (set! *val* (global-fetch i))
       (when (eq? *val* undefined-value)
         (signal-exception #t
           (list "Uninitialized global variable") ) ) ) )
    ((predefined)
     (let ((i (cdr kind)))
       (set! *val* (predefined-fetch i)) ) ) )
  (signal-exception #f (list "No such variable")) ) )
```

The function **shadowable-fetch** will use a statically known address to search for a dynamic address indicating where the variable is actually located. For that reason, we have to be able to decode all types of addresses, namely, **local** and **checked-local**, **global** and **predefined**, and even **shadowable**, as it can occur also. The function **variable-value-lookup** does that decoding. In a way, the form **import** corresponds to a kind of syntax transforming references to floating variables into calls to **eval/b**. Let's look again at our earlier example: the form[11] **(import (x y) env (list (set! x y) z))** is none other than this:

```
(list ((eval/b '(lambda (v) (set! x v)) env)
       (eval/b 'y env) )
      z )
```

Let's summarize all the types of bindings we've seen.

Lexical binding—the type implemented by **lambda**—allocates boxes to put values into them. The bound names make it possible to retrieve these boxes. Their extent is indefinite; that is, they go away only when they are no longer needed. The scope of these bindings is restricted to the body of the **lambda** form.

Dynamic binding—the type implemented by **dynamic-let**—associates a name with a value[12] throughout the duration of a given computation. The scope is unlimited during that computation. The association disappears afterward.

Quasi-static binding puts names on bindings that already exist. Moreover, since these are names of bindings that must be used again, introducing this kind of binding upsets α-conversion. Why? Because even in the absence of **procedure-environment**, reifying[13] an environment captures the names of bindings and prohibits us from knowing all the places where they might be used.

---

11. To avoid generating a program that includes a quotation that might not be legal, here we've generated a closure taking the value to assign.
12. As we implemented it, **dynamic-let** won't let you modify this association; there is no **dynamic-set!**. It's easy to get around this limitation by binding mutable data to this name.
13. We could improve **reflective-lambda** as explained earlier so that in addition it specified the kind of operations allowed on exported bindings by limiting them, for example, to read-only. Like [LF93], we could also allow free variables to be renamed for exportation.

8.7.6 Simplified Access to Environments

The preceding sections have shown that most ways of evaluation can be programmed explicitly except those that involve access to the global environment. The preceding digressions have given us an appreciation for first class environments and also lead us to generalize the functions `global-value` and `set-global-value!` into their counterparts `variable-value`, `set-variable-value!`, and `variable-defined?`.

The function `variable-value` takes a first class environment and looks for the value of a variable; `set-variable-value!` modifies it; `variable-defined?` tests whether it is present. They use the same functions to decode an address as the function `shadowable-fetch`: all take care not to use `compute-kind` directly as it "accepts" variables passed to it and creates them on the fly. We've left out `set-variable-value!`, but you can guess its definition easily enough.

```
(definitial variable-value
  (let* ((arity 2)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= (activation-frame-argument-length *val*) arity+1)
           (let ((name (activation-frame-argument *val* 0))
                 (env (activation-frame-argument *val* 1)) )
             (if (reified-environment? env)
                 (if (symbol? name)
                     (let* ((r (reified-environment-r env))
                            (sr (reified-environment-sr env))
                            (kind
                             (or (let ((var (assq name r)))
                                   (and (pair? var) (cadr var)) )
                                 (global-variable? g.current name)
                                 (global-variable? g.init name) ) ) )
                       (variable-value-lookup kind sr)
                       (set! *pc* (stack-pop)) )
                     (signal-exception
                      #f (list "Not a variable name" name) ) )
                 (signal-exception
                  #t (list "Not an environment" env) ) ) )
           (signal-exception #t (list "Incorrect arity"
                                      'variable-value )) ) ) ) ) )
(definitial variable-defined?
  (let* ((arity 2)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= (activation-frame-argument-length *val*) arity+1)
           (let ((name (activation-frame-argument *val* 0))
                 (env (activation-frame-argument *val* 1)) )
             (if (reified-environment? env)
                 (if (symbol? name)
                     (let* ((r (reified-environment-r env))
```

```
                (sr (reified-environment-sr env)) )
          (set! *val*
                (if (or (let ((var (assq name r)))
                          (and (pair? var) (cadr var)) )
                        (global-variable? g.current name)
                        (global-variable? g.init name) )
                    #t #f ) )
          (set! *pc* (stack-pop)) )
        (signal-exception
         #f (list "Not a variable name" name) ) )
      (signal-exception
       #t (list "Not an environment" env) ) ) )
  (signal-exception #t (list "Incorrect arity"
                             'variable-defined? )) ) ) ) ) )
```

The function **variable-defined?** is a function for inspecting first class environments. It determines whether or not a given variable occurs in a given environment. That question interests us since we can enrich such environments, but the question itself depends on the nature of the global environment: is it mutable or not? If the global environment is immutable, then **variable-defined?** is a function with a constant response: if a variable does not appear now in an immutable global environment, then it will never occur there. However, if the global environment can change, then it can be extended even while remaining equal to itself (as if we had an **enrich!** function) and thus **variable-defined?** might respond True at some point even after having responded False.

8.8 Reflective Interpreter

In the mid-eighties, there was a fashion for reflective interpreters, a fad that gave rise to a remarkable term: "reflective towers." Just imagine a marsh shrouded in mist and a rising tower with its summit lost in gray and cloudy skies—pure Rackham! In fact, the foundations of this mystical image are anchored in the experiments carried on around continuations, first class environments, and FEXPR of InterLisp (put to death by Kent Pitman in [Pit80]).

Well, who hasn't dreamed about inventing (or at least having available) a language where anything could be redefined, where our imagination could gallop unbridled, where we could play around in complete programming liberty without trammel nor hindrance? However, we pay for this dream with exasperatingly slow systems that are almost incompilable and plunge us into a world with few laws, hardly even any gravity.

At any rate, this section presents a small reflective interpreter in which few aspects are hidden or intangible. Many proposals for such an interpreter exist already, for example in [dRS84], [FW84], [Wan86], [DM88], [Baw88], [Que89], [IMY92], [JF92], and others. They are distinguished from each other by their particular aspects of reflection and by their implementations.

Reflective interpreters should support introspection, so they must offer the programmer a means of grabbing the computational context at any time. By "computational context," we mean the lexical environment and the continuation. To get

the right continuation, we already have `call/cc`. To get the right lexical environment, we'll take the form `(export)`, also known as `(the-environment)`, to reify the current lexical environment. Reification is one of the imperatives of reflection, but there are many ways of reifying, and how we choose to do it will affect the operations that we can carry out later.

As [Chr95] once said, "Puisqu'une fois la borne franchie, il n'est plus de limite." That is, once we've crossed the boundary, there are no more limits, so we must also authorize programs to define new special forms. InterLisp experimented with a mechanism already present in Lisp 1.5 under the name FEXPR. When such an object is invoked, instead of passing arguments to it, we give it the text of its parameters as well as the current lexical environment. Then it can evaluate them whatever way it wants to, or even more generally, it can manipulate them at will. For that purpose, we'll introduce the new special form `flambda` with the following syntax:

`(flambda (`*variables...*`)` *forms ...*`)`

The first variable receives the lexical environment at invocation, while the following variables receive the text of the call parameters. With such a mechanism, we can trivially write quotation like this:

```
(set! quote (flambda (r quotation) quotation))
```

A reflective interpreter must also provide means to modify itself (a real thrill, no doubt), so we'll make sure that functions implementing the interpreter are accessible to interpreted programs. The form `(the-environment)` insures this effect since it gives us access to the variables of the implementation as if they belonged to interpreted programs.

So here's a reflective interpreter, one weighing in at only[14] 1362 bytes. Thus you can see that it is not costly in terms of memory to get such an interpreter for ourselves in a library. It is written in the language that the byte-code compiler compiles.

```
(apply
 (lambda (make-toplevel make-flambda flambda? flambda-apply)
   (set! make-toplevel
         (lambda (prompt-in prompt-out)
           (call/cc
            (lambda (exit)
              (monitor (lambda (c b) (exit b))
                ((lambda (it extend error global-env
                          toplevel eval evlis eprogn reference )
                   (set! extend
                         (lambda (env names values)
                           (if (pair? names)
                               (if (pair? values)
                                   ((lambda (newenv)
                                      (begin
                                        (set-variable-value!
                                         (car names) newenv (car values) )
```

14. Once it's compiled, that is, because its text is only about 120 lines, 6000 characters, and 583 pairs.

```
                        (extend newenv (cdr names)
                                (cdr values) ) ) )
                    (enrich env (car names)) )
                   (error "Too few arguments" names) )
               (if (symbol? names)
                   ((lambda (newenv)
                      (begin
                        (set-variable-value!
                         names newenv values )
                        newenv ) )
                    (enrich env names) )
                   (if (null? names)
                       (if (null? values)
                           env
                           (error "Too much arguments"
                                  values ) )
                       env ) ) ) ) )
(set! error (lambda (msg hint)
              (exit (list msg hint)) ))
(set! toplevel
      (lambda (genv)
        (set! global-env genv)
        (display prompt-in)
        ((lambda (result)
           (set! it result)
           (display prompt-out)
           (display result)
           (newline) )
         ((lambda (e)
            (if (eof-object? e)
                (exit e)
                (eval e global-env) ) )
            (read) ) )
        (toplevel global-env) ) )
(set! eval
      (lambda (e r)
        (if (pair? e)
            ((lambda (f)
               (if (flambda? f)
                   (flambda-apply f r (cdr e))
                   (apply f (evlis (cdr e) r)) ) )
             (eval (car e) r) )
            (if (symbol? e) (reference e r) e) ) ) )
(set! evlis
      (lambda (e* r)
        (if (pair? e*)
            ((lambda (v)
               (cons v (evlis (cdr e*) r)) )
             (eval (car e*) r) )
            '() ) ) )
(set! eprogn
```

```
                    (lambda (e+ r)
                      (if (pair? (cdr e+))
                          (begin (eval (car e+) r)
                                 (eprogn (cdr e+) r) )
                          (eval (car e+) r) ) ) )
                (set! reference
                    (lambda (name r)
                      (if (variable-defined? name r)
                          (variable-value name r)
                          (if (variable-defined? name global-env)
                              (variable-value name global-env)
                              (error "No such variable" name) ) ) ) )
                ((lambda (quote if set! lambda flambda monitor)
                   (toplevel (the-environment)) )
                 (make-flambda
                  (lambda (r quotation) quotation) )
                 (make-flambda
                  (lambda (r condition then else)
                    (eval (if (eval condition r) then else) r) ) )
                 (make-flambda
                  (lambda (r name form)
                    ((lambda (v)
                       (if (variable-defined? name r)
                           (set-variable-value! name r v)
                           (if (variable-defined? name global-env)
                               (set-variable-value! name global-env v)
                               (error "No such variable" name) ) ))
                     (eval form r) ) ) )
                 (make-flambda
                  (lambda (r variables . body)
                    (lambda values
                      (eprogn body (extend r variables values)) ) ) )
                 (make-flambda
                  (lambda (r variables . body)
                    (make-flambda
                     (lambda (rr . parameters)
                       (eprogn body
                               (extend r variables
                                       (cons rr parameters) ) ) ) ) ) )
                 (make-flambda
                  (lambda (r handler . body)
                    (monitor (eval handler r)
                             (eprogn body r) ) ) ) ) )
               'it 'extend 'error 'global-env
               'toplevel 'eval 'evlis 'eprogn 'reference ) ) ) ) ) )
  (make-toplevel "?? " "== ") )
'make-toplevel
((lambda (flambda-tag)
   (list (lambda (behavior) (cons flambda-tag behavior))
         (lambda (o) (if (pair? o) (= (car o) flambda-tag) #f))
         (lambda (f r parms) (apply (cdr f) r parms)) ) )
```

```
98127634 ) )
```

As Julia Kristeva would say,[15] that definition is saturated with subtlety at least with respect to signs. You'll probably have to use this interpreter before you'll be able to believe in it.

The initial form (`apply` ...) creates four local variables. The last three all work on reflective abstractions or `flambda` forms; specifically, `make-flambda` encodes them; `flambda?` recognizes them; `flambda-apply` applies them. These objects have a very special call protocol, and for that reason, we must get acquainted with them. The first variable of the four, `make-toplevel` is initialized in the body so that it can make these four variables available to programmers. It starts an interaction loop with customizable prompts. In the beginning, these prompts are `??` and `==`.

This interaction loop captures its call continuation—the one where it will return in case of error or in case of a call to the function `exit`.[16] That continuation will also be accessible from programs. The interaction loop itself is protected by a form, `monitor`. [see p. 256] The loop then introduces and initializes an entire group of variables that will also be available for interpreted programs to share. The variable `it` is bound to the last value computed by the interaction loop. The function `extend`, of course, extends an environment with a list of variables and values; it also checks arity. The function `error` prints an error message and then terminates with a call to `exit`.

The function `toplevel` implements interaction in the usual way. The functions that accompany `eval` (namely, `evlis`, `eprogn`, and `reference`) are standard, too, except for the fact that they are accessible to programs. The function `eval` is simplicity itself. Either the expression to compute is a variable or an implicit quotation, or it's a form, in which case we evaluate the term in the function position. If that term is a reflective function, then we invoke it by passing it the current environment and the call parameters; if it's a normal function, we simply invoke it normally.

This flexibility is the characteristic that makes the language defined by this interpreter non-compilable. Let's assume that we've defined the following normal function:

```
(set! stammer (lambda (f x) (f f x)))
```

Now we can write two programs:

```
(stammer (lambda (ff yy) yy) 33)    → 33
(stammer (flambda (ff yy) yy) 33)   → x
```

According to their definitions, one executes its body; the other reifies one part of its body to provide that to its argument `f`. Consequently, we must compile every functional application twice to satisfy normal and reflective abstractions. However, since every program represented by a list is also possibly a functional application, we must also doubly compile forms beginning with the usual special forms, like `quote`, `if`, and others because they, too, could be redefined to do something else.

15. Well, she said something like that one Sunday morning on radio France Musique, 19 September 1993, while I was writing this.

16. Finally we have a Lisp from which we can exit! You can check for yourself that there is no such convenience in the definitions of COMMON LISP, Scheme, or even Dylan.

In short, there are practically no more invariants for the compiler to exploit to produce efficient code—thus leaving an open field for the joys of interpretation.

We define a few reflective functions just before starting the interaction loop because it is simpler to do so there than to program them. In that way, we define **quote**, **if**, **set!**, **lambda**, **flambda**, and **monitor**. Any others that are missing can be added explicitly, like this:

```
(set! global-env
      (enrich global-env 'begin 'the-environment) )
(set! the-environment
      (flambda (r) r) )
(set! begin
      (flambda (r . forms)
         (eprogn forms r) ) )
```

The variable **global-env** is bound to the reified environment containing all preceding functions and variables, including itself, and that's the strength of this procedure. Not only is the global environment provided to programers. but also they can modify it retroactively through the interpreter. The interpreter and the program share the same **global-env**. Assignment by either means leads to the same effect, to the same thrills and dangers. Because of this two-way binding, we can better express the earlier example, or even the following one, where we define a new special form: **when**.

```
(set! global-env (enrich global-env 'when))
(set! when
      (flambda (r condition . body)
        (if (eval condition r) (eprogn body r) #f) ) )
```

First of all, we extend the global environment with a new global variable, one that has not yet been initialized, of course. Immediately afterwards, we initialize it with an appropriate reflective function. Interestingly enough, this extension of the global environment is visible to the interpreter: assigning **when** with a reflective abstraction actually adds a new special form to the interpreter.

At this point, you're probably asking, "Just where is that mystical tower poking through the fog that was alluded to some time ago?" Programs can create new levels of interpretation with the function **make-toplevel**. They can also evaluate their own definition and thus achieve a truly astounding slowdown that way, speed that makes this possibility purely theoretical in fact. Auto-interpretation is a little different from reflection. We need to follow two hard rules to make this interpreter auto-interpretive. First, we must avoid using special forms when they have been redefined in order to avoid instability. That's why the body of the abstraction which binds **set!** and the others is reduced to **(toplevel (the-environment))**. Second, instances of **flambda** must be recognized by the two levels of interpretation. That's why we have used the label **flambda-tag**: it's unique, though it might be falsified.

A final possibility is that we can reify whatever we have just been doing (including the continuation and the environment), think about whatever we have just been doing, imagine what we should have done instead, and do it. According to the way of programming that we've adopted for reification, there may be new and

exciting possibilities for introspection, like scutinizing the environment or control blocks in the continuation, as in [Que93a]. In fact, these kinds of introspection are the reason this type of interpreter is known as "reflective."

The Form `define`

Interestingly enough, the preceding reflective interpreter supports an operational definition of a highly complex form, **`define`** in Scheme. As we've already explained, **`define`** is a special, special form: it behaves like a declaration doubled by an assignment. It's like assignment because after execution, the variable will have a well defined value. It's like a declaration because it introduces a new variable as soon as the text mentioning **`define`** is prepared for execution. This declaration changes the global environment. All these aspects are highlighted in the following definition of the special form, **`define`**, but we've excluded syntactic aspects that **`define`** also has; those syntactic aspects let **`define`** accept variations like **`(define (foo x) (define (bar) ... ) ... )`** and participate in various situations known as internal **`define`**s. (You can measure how complex a special form **`define`** is.)

```
(set! global-env (enrich global-env 'define))
(set! define (flambda (r name form)
               (if (variable-defined? name r)
                   (set-variable-value! name r (eval form r))
                   ((lambda (rr)
                      (set! global-env rr)
                      (set-variable-value! name rr (eval form rr)) )
                    (enrich r name) ) ) ))
```

First of all, **`define`** tests whether the variable to define already belongs to the global environment. In that case, the definition is merely an assignment, according to R4RS §5.2.1. In the opposite case, a new binding is created inside the global environment, the global environment is updated to include this new binding, and the value to assign is computed in this new environment, in order to support recursions.

8.9 Conclusions

This chapter presented various aspects of explicit evaluation, aspects that appear as special forms or as functions. According to the qualities that we want to keep in a language, we might prefer forms stripped of all the "fat" of evaluation like first class environments or like quasi-static scope. You can clearly see here that the art of designing languages is quite subtle and offers an immense range of solutions.

This chapter also clearly shows how fortunate a choice Lisp made (or, more precisely, the choice its inspired designer, John McCarthy, made) by reducing how much coding and decoding would be needed and by mixing the usual levels of language and metalanguage to increase the field for experiment so greatly.

8.10 Exercises

Exercise 8.1 : Why doesn't the function `variables-list?` test whether the list of variables is non-cyclic?

Exercise 8.2 : The special form `eval/ce` compiles on the fly the expression given to it. That's too bad if you want to evaluate the same expression several times in succession. Think up a way to correct this problem.

Exercise 8.3 : Improve the definition of `eval/at` in terms of `eval/ce` in order to get rid of the inadvertantly captured variable. Hint: use `gensym` if you want.

Exercise 8.4 : Can a user define `variable-defined?` him- or herself? Why or why not?

Exercise 8.5 : Make the reflective interpreter run in pure Scheme.

Recommended Reading

The article [dR87] remarkably reveals just how reflective Lisp is. [Mul92] offers algebraic semantics for the reflective aspects of Lisp. For reflection in general, you should look at recent work by [JF92] and [IMY92].

9
Macros: Their Use & Abuse

IGNORED, abused, unjustly criticized, insufficiently justified (theoretically), macros are no less than one of the fundamental bases of Lisp and have contributed significantly to the longevity of the language itself. While functions abstract computations and objects abstract data, macros abstract the structure of programs. This chapter presents macros and explores the problems they pose. By far one of the least studied topics in Lisp, there is enormous variation in macros in the implementation of Lisp or Scheme. Though this chapter contains few programs, it tries to sweep through the domain where these little known beings—macros—have evolved.

Invented by Timothy P. Hart [SG93] in 1963 shortly after the publication of the Lisp 1.5 reference manual, macros turned out to be one of the essential ingredients of Lisp. Macros authorize programmers to imagine and implement the language appropriate to their own problem. Like mathematics, where we continually invent new abbreviations appropriate for expressing new concepts, dialects of Lisp extend the language by means of new syntactic constructions. Don't get me wrong: I'm not talking about augmenting the language by means of a library of functions covering a particular domain. A Lisp with a library of graphic functions for drawing is still Lisp and no more than Lisp. The kind of extensions I'm talking about introduce new syntactic forms that actually increase the programmer's power.

Extending a language means introducing new notation that announces that we can write X when we want to signify Y. Then every time programmers write X, they could have written Y directly, if they were less lazy. However, they are intelligently lazy and thus use a simple form to eliminate senseless details so that those details no longer encumber their thoughts. Many mathematical concepts become usable only after someone invents a suitable notation to express them. To insure greater flexibility, the rule about abbreviations usually exploits parameters. In fact, when we write $X(t_1, \ldots, t_n)$, we intend $Y(t_1, \ldots, t_n)$. Macros are not just a hack, but a highly useful abstraction technique, working on programs by means of their representation.

Most imperative languages have only a fixed number of special forms. You can usually find a `while` loop in one, and perhaps an `until` loop, but if you need a new kind of loop, for example, it's generally not possible to add one. In Lisp, in contrast, all you have to do is introduce the new notation. Here's an example of what we

mean: every time we write (**repeat** :**while** *p* :**unless** *q* :**do** *body*...), in fact we mean this:

```
(let loop ()
  (if p (begin (if (not q) (begin body...))
               (loop) )) )
```

This example is deliberately extravagant: it takes extra keywords (recognizable by the colon prefixing each one) although customary use in Scheme is rather to suppress such syntactic noise. **[see Ex. 9.1]** Also, the example introduces a local variable, **loop**, that might hide a variable of the same name that *p* or *q* or *body* might want to refer to. Real loop-lovers should also look at the Mount Everest of loops (the macro **loop** for which the popular implementation takes tens of pages) defined in [Ste90, Chapter 26].

Unfortunately, like many other concepts, especially the most advanced or the most subtle, macros can run amuck. The goal of this chapter is to unravel their problems and show off their beauties. To do so, we'll logically reconstruct macros so that we can survey their problems and the roots of their variations.

9.1 Preparation for Macros

The evaluators that evolved in the immediately preceding chapters distinguish two phases as they handle programs: *preparation* (the term used in IS-Lisp) followed by *execution*. Evaluation, as in fast interpretation, **[see p. 183]**, can thus be seen as (**run** (**prepare** *expression*)). That way of looking at things was quite apparent not only in the fast interpreter but also in the byte-code compiler where prepared expressions were successively a tree of thunks **[see p. 223]** or a byte vector. At worst, in all the early interpreters we developed, we could assimilate preparation with the identity.

Preparation itself can be divided into many phases. For example, realizing that the expressions to evaluate are initially character strings, many Lisps offer the idea of macro-characters that influence the syntactic analysis of a string during its transformation into an S-expression. The outstanding example of a macro-character is the quote: when it is read, it reads the expression that follows and inserts that expression into a form prefixed by the symbol **quote**. We could reproduce that process as (**list** (**quote quote**) (**read**)).

So that the Lisp reader can be influenced, the **read** function to a certain degree makes it possible to adapt the language to special needs. That's the case for Bigloo [SW94]. When it compiles a program written in Caml Light, it chooses the appropriate read function; the only constraint is that a compilable S-expression will be returned. In fact, this read function is the front-end of the Caml Light compiler [LW93].

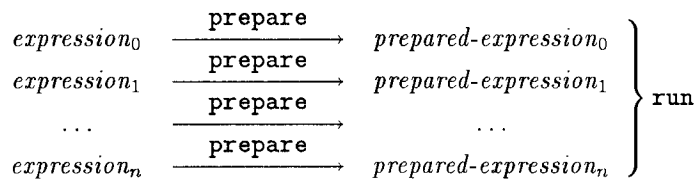
We won't dwell any longer, though, on the subject of macro-characters as they are highly dependent on the algorithm for reading S-expressions.

The preparation phase is often assimilated to a compilation phase of varying complexity. The important point is to maintain a strict separation between this phase and the following one (that is, execution) to clarify their relation, to minimize their common interface, and above all to insure that experiments can be repeated.

This imperative gives rise to two different ways of thinking about preparation: in terms of *multiple worlds* or in terms of a *unique world.*

9.1.1 Multiple Worlds

With multiple worlds, an expression is prepared by producing a result often stored in a file (conventionally, a `.o` or `.fasl` file). Such a file is executed by a distinct process which sometimes has a facility for gathering expressions prepared separately (that is, *linking*). Most conventional languages work in this mode, based on the idea of independent or separate compilation. In this way, it's possible to factor preparation and to manage name spaces better by means of import and export directives. We speak of this way of doing things as "multiple worlds" because the expression that is being prepared is the only means of communication between the expression-preparer and the ultimate evaluator. Those two processes live in distinct worlds. No effects in the first world is perceptible in the second.



9.1.2 Unique World

In contrast to multiple worlds, the hypothesis of a unique world stipulates that the center of the universe is the toplevel loop and that thus all computations started from there cohabit in the same memory where we can neither limit nor control communication by altering globally visible resources (such as global variables, property lists, etc.). In the unique world, the interaction loop ties together the reading of an expression, its preparation, and then its evaluation. This interaction loop plays the role of a command interpreter (a veritable *shell*) but nevertheless makes it possible to prepare expressions without evaluating them immediately after. In many systems, that's exactly what the function `compile-file` does: it takes the name of a file containing a sequence of expressions, prepares them, and produces a file of prepared expressions. The evaluation of a prepared file can also occur from the interaction loop by means of the function `load` or one of its derivatives. You see then that you could factor the preparation of an expression and thus interlace preparation and evaluation.

The idea of a unique world is not so bad. In fact, it's only a reflection of the one-world mentality of an operating system like UN⋆X, where the user counts on a certain internal state depending simultaneously on the file system, local variables, global variables, aliases, and so forth that his or her favorite command interpreter (`sh`, `tcsh`, etc.) provides. But again, the point is to insure, as simply as possible, that experiments can be repeated.

When we build software, it's a good idea to have a reliable method for getting an executable from it. We want any two reconstructions starting from the same source to end up in the same result. That's just a basic intellectual premise. Without too much difficulty, it is insured by the idea of multiple worlds because there we can

easily control the information submitted as input. It's a bit harder with the unique world hypothesis because we have difficulty mastering its internal state[1] and its hidden communications. Instead of being re-initialized for every preparation, the preparer (that is, the compiler) stays the same and is modified little by little since it works in a memory image that is constantly being enriched and always less under our control.

In short, the preparation of a program has to be a process that we control completely.

9.2 Macro Expansion

We need a way of abbreviating, but one that belongs strictly to preparation. For that reason, we might split preparation into two parts: first, *macro expansion*, followed by preparation itself. On this new ground, there are two warring factions: *exogenous* macro expansion and *endogenous* macro expansion. We can prepare an expression by this succession: `(really-prepare (macroexpand expression))`. We're interested only in the *macro expander*, that is, the function that implements the process of macro expansion: `macroexpand`. Where does it come from?

9.2.1 Exogenous Mode

The exogenous school of macro expansion, as defined in [QP91b, DPS94b], stipulates that the function `macroexpand` is provided *independently* of the expression to prepare, for example, by preparation directives.[2] Thus the function `prepare` looks something like this:

```
(define (prepare expression directives)
  (let ((macroexpand (generate-macroexpand directives)))
    (really-prepare (macroexpand expression)) ) )
```

Several implementations exist. We'll illustrate them with the byte-code compiler we presented earlier, [see p. 262]. We'll assume that the main functionalities of this compilation chain are: `run` to run an application and `build-application` to construct an application. We'll also assume that `compile.so` is the executable corresponding to the entire compiler. In the examples that follow, the macro expansion directive simply mentions the name of the executable in charge of the macro expansion.

A macro expansion might occur as a cascade, like the top of Figure 9.1. In that case, a command like "`compile file.scm expand.so`" (where the file to compile is specified in the first position and the macro expander in the second position) corresponds in pseudo-UN⋆X to this:

```
run expand.so < file.scm | run compile.so > file.so
```

A macro expansion might also occur through the synthesis of a new compiler (here, `tmp.so`), like the bottom of Figure 9.1. Then the command "`compile file.scm expand.so`" is analogous to this:

1. Who knows, for example, all the environment variables that `printenv` reveals or all the `.*rc` files on which our comfort depends?
2. These directives may, for instance, be found in the first S-expression of a file.

```
( build-application compile.so expand.so -o tmp.so ;
  run tmp.so ) < file.scm > file.so
```

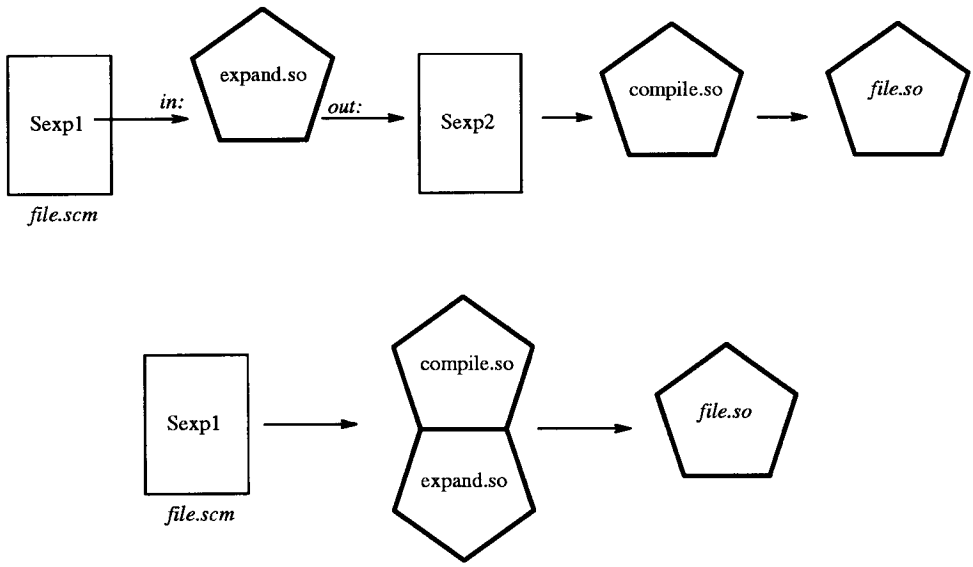


Figure 9.1 Two examples of exogenous mode (with the byte-code compiler); pentagons represent compiled modules.

Among the hidden problems, we still have to define a protocol for exchanging information between the preparer and the macro expander. The macro expander must receive the expression to expand; it might receive the expression as a value or as the name of a file to read; it must then return the result to its caller; at that point, it might return an S-expression or the name of a file in which the result has been written. Passing information by files is not absurd. In fact, it's the technique used by the C compiler (`cc` and `cpp`). As a technique, it prevents hidden communication; that is, every communication is evident. In contrast, passing information by S-expressions means that the preparer must know how to call the macro expander dynamically, either to evaluate or to load, by `load`, since they are both in the same memory space.

Rather than impose a monolithic macro expander, we might build it from smaller elements. We might thus want to compose it of various abbreviations. That comes down to extending the directive language so that we can specify all the abbreviations. Of course, doing that means that the macros must be composable, and that point in turn implies that we must define protocols to insure such things as the associativity and commutativity of macro definitions.

In consequence, the idea of exogeny associates an independently built macro expander with a program. If we organize things in this way, we can be sure that experiments can be repeated; we can also be sure that remains from macro expansions do not stay around in the prepared programs. Besides achieving the perfect separation we wanted between macro expansion and evaluation, this solution puts

no limits on our imagination since any macro expander is legal as long as it leads to an S-expression. We can even use `cpp`, `m4`, or `perl` for the job! However, we're sure that to specify an algorithm for transforming S-expressions, Lisp is the most adequate choice. You see from these observations that inside the exogenous vision, there is no connection between the language of the macro expander and the language that is being prepared. The macro expander only has to search for the sites to expand since the definitions of macros are not in the text to prepare.

9.2.2 Endogenous Mode

The endogenous mode depends on directives. It insists that all information necessary for the expansion must be found *in* the expression to prepare. In other words, the preparation function is defined like this:

```
(define (prepare expression)
  (really-prepare (macroexpand expression)) )
```

The fundamental difference between exogeny and endogeny is that with endogeny, the algorithm for macro expansion pre-exists, so we can only parameterize it and that only within limits. The predefined macro expansion algorithm thus now has the double duty of finding the definitions of macros as well as the sites where they are used—not so simple as before. The definitions of abbreviations are thus passed as S-expressions that often begin with a keyword like `define-macro` or `define-syntax`. That has to be converted into an expander, that is, a function whose role is to transform an abbreviation into the text that it represents. Thus the macro expander contains the equivalent of a function for converting a text into a program! There's the stroke of genius: rather than invent a special language for macros, we simply use the function `eval`. In other words, the definition language for macros is Lisp!

Of course, that stroke of genius is also the source of problems. Macros bring up again an old and subtle technique that was long ago assimilated as part of black magic: macrology. In fact, it's precisely this identification between the languages that fogs up our perception of what macros really are. For a long time, they were seen as real functions except that their calling protocol was a little strange because their arguments were not evaluated but their results were. That model, while it was convenient in an interpreted unique world, has been abandoned since we've made such progress in compiler design and for reasons cited in the article by Kent Pitman [Pit80]. Now we think of a macro as a process whose social role is to convert texts filled with abbreviations into new texts stripped of abbreviations.

Our choice of Lisp as the language for defining macros is resisted by those who restrict this language to filtering. Such are the macros proposed in the appendix of R^{4}RS. By doing that, they lose expressiveness since certain transformations can no longer be expressed, but they gain clarity in the simpler macros that are nevertheless the most common ones that we write. As an example, programs associated with this book at the time of publication contain 63 macro definitions using `define-syntax` as opposed to only 5 using the more liberal `define-abbreviation`.[3] The last three of those five uses define fundamental macros of ME-

3. We chose this exotic name to avoid conflict with various semantics conferred on `define-macro` by the different Lisp and Scheme evaluators.

ROONET (`define-class`, `define-generic`, and `define-method`); they couldn't be programmed with `define-syntax`.

In short, whether we impose a procedure for macro expansion or whether we decide to exploit parameters for an existing mechanism (at the price of incorporating a local evaluator for macro expansion), both models favor Lisp as the definition language for macros. In contrast to the exogenous mode, the endogenous mode requires an evaluator inside the macro expander; that's its characteristic trait. However, you must not confuse the language for writing macros with the language that is being prepared: they are related but not the same. When old fashioned manuals describe macros as things that work by double evaluation, they sin by omission since they don't mention that the two evaluations involve two different evaluators.

Figure 9.2 Exogenous (left) and endogenous (right) macroexpansions

9.3 Calling Macros

The macro expander has to find the places where the abbreviations to replace are located. Some languages, for example [Car93], already have elaborate syntactic analyzers that we can extend with new grammar productions for the syntactic extensions we want. That can be as simple as syntactic declarations indicating whether an operator is unary or binary, or prefix, postfix, infix. Sometimes we can also add an indication of precedence.

In Lisp, the representation of S-expressions favors the extraction of `car` from forms, so the following scheme is widely distributed: a list where the `car` is a keyword is a call to the macro of the same name. This scheme has many virtues: it is extendable, simple, and uniform. An abbreviation (that is, a macro) has a name associated with a function (an *expander*). The algorithm for macro expansion is thus quite simple: it runs recursively through the expression to handle, and when it identifies a call to a macro, it invokes the associated expander, providing the expander the S-expression that set things off. In the `cdr` of the S-expression, the expander can find whatever parameters for the expansion have been delivered to it.

The idea of a macro symbol comes in here, too, as in COMMON LISP, where we can connect the activation of an abbreviation with the ocurrence of a symbol. This offers us a lighter form, stripped of superfluous parentheses, when we want to associate some treatment with something that superficially resembles a variable.

A macro in that sense is thus just an association between an expansion function and a name matching a condition of activation. It's a kind of binding in what looks like a new name space reserved for macros. The macro is not exactly a binding, nor the expander, nor the activation condition, but everything all three imply inside

the macro expander. As we do in any conveniently managed name space, we want to be able to define both global and local macros there: `define-abbreviation` and `let-abbreviation` introduce such macros.

The purpose of the function for macro expansion is to convert an S-expression into a program that can be assimilated. The consequence is that the search for sites where macros are called occurs in an S-expression, not in a program. Of course, this S-expression has some parts that are already programs. Equally certain, too, a site of a call resembles a functional application but basically this is only an S-expression, and nothing prevents a macro from showing bad taste. Accordingly, the S-expression `(foo . 5)` could be defined in an ugly way as an abbreviation for `(vector-ref foo 5)`!

The problem of the macro expander is thus to survey an S-expression, a grammatically soft structure with respect to the grammar of programs. That poses certain problems of precedence between macros and lexical behavior. So if `bar` is a macro and `lambda` is not one, then is the expression `(bar 34)` appearing in `((lambda (bar) (bar 34)) ... )` a site where the macro `bar` is being used? The problem arises because of the presence of the local variable, also named `bar`. Is it hiding the macro `bar`? R^{4}RS takes a position favoring the respect for lexicality. The same confusion also exists for quotations. Does `(quote (bar 34))` contain a site where the macro `bar` is being used?

It is unfortunately also necessary to look into the algorithm for expansion so that we can see, when it scans an S-expression, how it finds the positions in which it might find the sites where macros are called. For example, can we write `(let (foo) ... )` in a context where `(foo)` is a macro call generating a set of bindings? A great many other examples exist, like `(cond (foo) ... )` or `(case key (foo) ... )`. In Lisp, good taste demands that we respect as much as possible the grammar of forms in a way that confuses the difference between functions and macros. Without more information, `(bar ...)` might be a call to the function `bar` or the site of a call to the macro of the same name.

9.4 Expanders

What contract does an expander offer? At least two solutions co-exist.

In *classic* mode, the result delivered by the expander is considered as an S-expression which can yet again include abbreviations. For that reason, the result is re-expanded until it no longer contains any abbreviations. If `find-expander` is the function which associates an expander with the name of a macro, then the expansion of a macro call such as `(foo . `*expr*`)` can be defined like this:

```
(macroexpand '(foo . expr)) ⇝
    (macroexpand ((find-expander 'foo) '(foo . expr)))
```

Another, more complicated, mode also exists. It's known as *Expansion Passing Style* or EPS in [DFH88]. There the contract is that the expander must return the expression completely stripped of abbreviations. Thus there is no `macroexpand` waiting on the return of the call to the expander. The difficulty is that the expander must recursively expand the subterms making up the expression that it generates. It can do that only if it, too, knows the way of invoking macro expansion. One

solution is make sure that macro expanders have access to the global variable **macroexpand**, whose value is the function for macro expansion. Another, more clever solution, takes into account the need for local macros and modifies the calling protocol for expanders so that they take as a second argument the current function for macro expansion. In Lisp-like terms, we thus have this:

```
(macroexpand '(foo . expr)) ⇝
    ((find-expander 'foo) '(foo . expr) macroexpand)
```

Those two systems—classic versus EPS—are not equivalent. Indeed, EPS is clearly superior in that it can program the former. A major difference in their behavior concerns the need for local macros. Let's assume that we want to introduce an abbreviation locally. In EPS, we can extend the function **macroexpand** to handle this abbreviation locally on subterms. In classic mode, for example in COMMON LISP, the same request is treated this way: a special macro **macrolet** modifies the internal state of the macro expander to add local macros there; then it expands its body; then it remodifies the internal state to remove those macros that it previously introduced there. The macro **macrolet** is primitive in the sense that we cannot create it if by chance we don't have it. Likewise, in classic mode, it is not possible to de-activate a macro external to **macrolet** (it is necessarily visible or shadowed by a local macro) although in EPS, we can locally introduce syntax arbitrarily different locally from the surrounding syntax. All we have to do is recursively use another function on the subterms—a function other than **macroexpand**. In [DFH88], there are many examples of tracing and currying in this way.

Most abbreviations have local effects, that is, they lead only to a substitution of one particular text for another. It's important that EPS lets us access the current expansion function as it does because then we can simply program more complex transformations than those supported by the classic mode without having to program a code walker. For example, think of the program transformation that introduces boxes [see p. 114]. In EPS, it's possible to handle them by carrying out a preliminary inspection of the expression to find local mutable variables and then a second walk through to transform all the references to these variables, whether read- or write-references.

However, EPS is not all powerful; there are transformations beyond its scope. [see p. 141] Extracting quotations is not a local transformation beause, when we encounter a quotation, extraction obliges us to transform the quotation into a reference to a global variable. Up to that point, there's no problem, but we must also create that global variable (we do by inserting a **define** form in the right place) and that is hardly a local text effect! One solution would be to enclose the entire program in a macro (let's say, **with-quotations-extracted**) which would return a sequence of definitions of quotations followed by the transformed program.

Other macros might need to create global variables, too, like the macro **define-class** in MEROONET. Syntactically, it can appear inside a **let** form. Its contract is to create a global variable containing the object representing the class. This problem is not generally handled by macro systems, so it obliges **define-class** to be a special form or a macro exploiting magic functions, that is, known only by implementations.

Since EPS is so interesting, you might ask why it's not used more often. First,

because of the complexity of the model, but the main reason (as we've already mentioned) is that, on the whole, macros are simple so making them all-powerful only slows down their expansion. In effect, one interesting property of the classic mode is that we never get back to the previously expanded expressions: when the macro being expanded does not begin by a macro keyword, then it's a functional application which we'll never see again. Macro expansion occurs in one sole linear pass, although the result of an expansion in EPS can always be taken up again by an embedding expander, and thus it tends to cause superfluous re-expansions.

9.5 Acceptability of an Expanded Macro

By contract, the result of a macro expansion must be a program ready to be prepared. It shouldn't contain any more abbreviations; that is, it should appear exactly like it would have been written directly. There are a few pitfalls to avoid in getting there. First, since the macro expansion is a computation, there is a possibility that it might not terminate, and as a consequence, the preparation phase would not terminate either. It doesn't happen often that a compiler loops, but when it does, it's the price we pay to have a powerful macro system that does not limit the kind of computations we can do with it.

An easy way to cause looping is to re-introduce the expression to the macro expander in the expanded macro. That could easily happen if we defined the macro **while** like this:

LOOP

```
(define-abbreviation (while condition . body)
  `(if ,condition (begin (begin . ,body) (while ,condition . ,body))) )
```

Re-introducing the same text that we are in the process of expanding and putting it into the result of the macro expansion provokes an endless expansion in every sense of the phrase, especially if we are in the classic mode of expansion.

The same error can occur in a way that is even more surreptitious in COMMON LISP because of the keyword **&whole**. That keyword lets the macro recover the whole calling form. In the following definition, the result of the macro expansion contains the original expression.

COMMON LISP

```
(defmacro while (&whole call)
  (let ((condition (cadr call))
        (body      (cddr call)) )
    `(if ,condition (begin (begin . ,body) ,call)) ) )
```

Many interpreters macro-expand S-expressions on the fly as they are received. To avoid expanding the same thing again and again, they could adopt a technique of *memo-izing* or *displacing* macros. That strategy consists of physically replacing the S-expression that set off the macro expansion—replacing it by the expansion that yields the macro. In that case, the preceding **while** macro would have built a cyclic structure, posing no problem for interpretation but making the compiler loop because the compiler expects to handle only DAGs (directed acyclic graphs, that is trees possibly with shared branches) [Que92a]. We can show you that variation by rewriting the **while** macro like this:

COMMON LISP

```
(defmacro while (&whole call)
  (let ((condition (cadr call))
```

```
    (body      (cddr call)) )
(setf (car call) 'if)
(setf (cdr call) `(,condition (begin (begin . ,body) ,call)))
call ) )
```

What we've just shown you about programs can also occur with quotations; they, too, can be cyclic and thus make certain compilers loop. [see p. 140]

The second pitfall, even nastier than the first, is that the expanded macro can contain values intruding from the macro expansion. The moral contract of the programmer is that the result of the macro expansion should be a program that he or she could have written directly. That implies that the program necessarily has a writable form. Scheme thus insists that quotations must be formulated with an external syntax in order to prevent the quotation of just any values.

Let's look at a particularly torturous example of a quotation with a non-writable value. The following macro builds just that by inserting a continuation into the quoted value, like this:

```
(define-abbreviation (incredible x)                    BAD TASTE
  (call/cc (lambda (k) `(quote (,k ,x)))) )
```

What meaning should we give that gibberish? Let's reconsider a few of the hypotheses we've already mentioned. Writing the expanded macro in a file is impossible since we don't have a standard way of transcribing continuations. What does the invocation of a continuation taken from a different process mean anyway? Or, in C terms, what does it mean during the execution of `a.out` to invoke a continuation captured by `cpp`? In the absence of any agreement about what that means, we should just say, "No!"

That example used a continuation to highlight the strangeness of the situation. All the same, we will avoid including any value that has no written representation; likewise, we won't include primitives, closures, nor the input and output ports. Consequently, we will no longer write `(',(lambda (y) car) x)` nor `'(',car x)` nor `(f ',(current-output-port))`, even if in an interpreted world that would not be an error.

In short, we'll repeat the golden rule of macros: never generate a program that you could not write directly.

9.6 Defining Macros

Various forms declare global macros (with a scope that we'll analyze in the next section) and local macros. They are `define-syntax`, `letrec-syntax`, and `let-syntax` in Scheme; `defmacro` and `macrolet` in COMMON LISP; `define-abbreviation` and `let-abbreviation` in this book.

To define a macro, we build a function (the expander), and then register that expander with an appropriate name. The expander is written in the language of macros, an instance of Lisp. Now we'll analyze its consequences in the various worlds and modes that we mentioned earlier.

9.6.1 Multiple Worlds

Remember that with multiple worlds, macro expansion is a process that occurs in a memory space separated from the final evaluation.

Endogenous Mode

According to the endogenous school, the text defining a macro (in our case, the form **define-abbreviation**) synthesizes an expander on the fly by means of the evaluator implementing the macro language. In that case, the keyword **define-abbreviation** can be none other than the syntactic marker that the macro expander is searching for. The following definition presents a very naive macro expander that implements this endogenous strategy.

```
(define *macros* '())
(define (install-macro! name expander)
  (set! *macros* (cons (cons name expander) *macros*)) )
(define (naive-endogeneous-macroexpander exps)
  (define (macro-definition? exp)
    (and (pair? exp)
         (eq? (car exp) 'define-abbreviation) ) )
  (if (pair? exps)
      (if (macro-definition? (car exps))
          (let* ((def       (car exps))
                 (name      (car (cadr def)))
                 (variables (cdr (cadr def)))
                 (body      (cddr def)) )
            (install-macro! name (macro-eval
                                  `(lambda ,variables . ,body) ))
            (naive-endogeneous-macroexpander (cdr exps)) )
          (let ((exp (expand-expression (car exps) *macros*)))
            (cons exp (naive-endogeneous-macroexpander (cdr exps))) ) )
      '() ) )
```

That macro expander takes a sequence of expressions (the set of expressions from a file to compile) and consults the global variable ***macros***, which contains the current macros in the form of an association-list. One by one, the expressions are expanded by the subfunction **expand-expression** with the current macros. When the definition of a macro is encountered, it is handled specially, on the fly, like an evaluation inside the macro expansion. The evaluation function is represented by **macro-eval**; it might be different from native **eval**. Indeed, **macro-eval** implements the macro language, whereas **eval** implements the language into which we're expanding macros. This looks a little like cross-compilation. Once the expander has been built, it is inserted by **install-macro!** in the list of current macros. You can imagine the world of difference between the definitions of macros and the other expressions which are only expanded. The following correct example illustrates that idea.

---

si/chap9b.scm

```
(define (fact1 n) (if (= n 0) 1 (* n (fact1 (- n 1)))))
(define-abbreviation (factorial n)
  (define (fact2 n) (if (= n 1) 1 (* n (fact2 (- n 1)))))
  (if (and (integer? n) (> n 0)) (fact2 n) '(fact1 ,n)) )
(define (some-facts)
  (list (factorial 5) (factorial (+ 3 2))) )
```

It wouldn't be healthy to confuse `fact1` and `fact2` in endogenous mode because the definition of `fact1` is merely expanded and not evaluated; it doesn't even exist yet when the macro `factorial` is defined. For that reason, the macro uses its own version—`fact2`—for its own needs. Thus expanding those three expressions leads to this:

si/chap9b.escm

```
(define (fact1 n) (if (= n 0) 1 (* n (fact1 (- n 1)))))
(define (some-facts) (list 120 (fact1 (+ 3 2))))
```

We could make that example more complicated by renaming `fact1` and `fact2` simply as `fact`. Then mentally we would have to keep track of which occurrence of the word `fact` inside the definition of `factorial` refers to which. We would also have to determine that the first occurrence refers to the value of the variable `fact` local to the macro at the time of expansion, while the second refers to the global variable `fact` at the time of execution.

There are other variations within endogenous mode. Some macro expanders make a first pass to extract the macro definitions from the set of expressions to expand. Those are then defined and serve to expand the rest of the module. This submode poses problems when macro calls define new macros because the macro expander risks not seeing them. Moreover, the fact that macros can be applied even before they are defined is disorienting enough. There again, left to right order seems mentally well adapted.

Another important variation—one that we'll see again later—is to make `define-abbreviation` a predefined macro.

Exogenous Mode

In exogenous mode, macros are no longer defined in expressions to prepare, but rather in expressions that define the macro expander. We will assume that the macro expander is modular; that is, that we can define independent macros in a separate way. The best way to define such macros is, of course, to have a macro to do so. We'll use the keyword `define-abbreviation`, and here's its metacircular definition:

```
(define-abbreviation (define-abbreviation call . body)
  '(install-macro! ',(car call) (lambda ,(cdr call) . ,body)) )
```

When the macro expander is specified by an appropriate directive, its role is to add to itself the macro just defined. We'll thus assume that the macro expansion

directives specify an expansion engine that is enriched by calls to `install-macro!`. Now mastery of macro expansion is completely different since macros are defined in some modules and used in others. Let's assume that we have a preliminary module with the following contents:

si/chap9c.scm

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
(define-abbreviation (factorial n)
  (if (and (integer? n) (> n 0)) (fact n) '(fact ,n)) )
```

The expansion of that module would look like this:

si/chap9c.escm

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
(install-macro! 'factorial
   (lambda (n) (if (and (integer? n) (> n 0)) (fact n) '(fact ,n))) )
```

Now let's suppose we want to use the macro `factorial` for macro expansion of a module containing the definition of the function `some-facts`, like this:

si/chap9d.scm

```
(define (some-facts)
  (list (factorial 5) (factorial (+ 3 2))) )
```

We would get these results:

si/chap9d.escm

```
(define (some-facts)
  (list 120 (fact (+ 3 2))) )
```

The first occurrence of `fact` in the macro `factorial` poses no problem; it's just a reference to the global variable in the same module `si/chap9c.scm`. That's obvious if we look at its macro expansion in `si/chap9c.escm`. In contrast, the second occurrence of `fact` refers to the variable `fact` as it will exist at execution time in the module `si/chap9d.escm`. That reference is still free (in the sense of unbound) in the expanded macro, and it might very well happen that there will be no variable `fact` at execution time!

An easy solution to that problem is to add (by linking or by cutting and pasting) a module to `si/chap9d.scm`—a module from among those we have on hand: `si/chap9c.scm` defines a global variable that meets our needs. But in doing so, we will also have imported the definition of a macro, `factorial`, found in the same

module, there only for the evaluation of `some-facts`, and which may introduce new errors since it invokes the function `install-macro!` which has no purpose in the generated application.

The second solution is to refine our understanding of the dependences that exist between the macros and their execution libraries, especially, *especially* the various moments of computation. The art of macros is complex because it requires a mastery of time. Let's look again at the example of `fact1`, `fact2`, and `factorial`. When we discover a site where `factorial` is used, the macro needs the function `fact2` for expansion. In contrast, its result (the macro expanded expression) needs the function `fact1` for execution. We'll say that `fact1` belongs to the *execution library* of the macro `factorial`, whereas `fact2` belongs to the *expansion library* of `factorial`. The extents of these two libraries are completely unrelated to each other. Indeed, the expansion library is needed only during macro expansion, whereas the execution library is useful only for the evaluation of the expanded macro.

Thus in exogenous mode, one solution is to define macros and their expansion library in one module, `si/libexp`, and then define their execution library in a separate module, `si/librun`. [see **Ex. 9.5**] Still working on the factorial, here's what we would write in one module:

si/libexp.scm

```
(define (fact2 n) (if (= n 0) 1 (* n (fact2 (- n 1)))))
(define-abbreviation (factorial n)
  (if (and (integer? n) (> n 0)) (fact2 n) `(fact1 ,n)) )
```

and in the other module:

si/librun.scm

```
(define (fact1 n) (if (= n 0) 1 (* n (fact1 (- n 1)))))
```

When a directive mentions the use of the macro `factorial`, the macro expander will load the module `si/libexp.scm`, and the directive language will register the fact that at execution time the library `si/librun.scm` must be linked to the application. In that way, we will be able to keep only the necessary resources at execution time, as in [DPS94b]. Figure 9.3 recapitulates these operations. It shows how we first construct the compiler adapted to compilation of the program, and then how we get the little autonomous executable that we are entitled to expect.

9.6.2 Unique World

After the preceding section, you might think that hypothesizing a unique world (rather than multiple worlds) would resolve all problems. Not so at all!

In a unique world, the evaluator reads expressions, expands the macros in them, prepares the expanded macros, and evaluates the result of the preparation. The

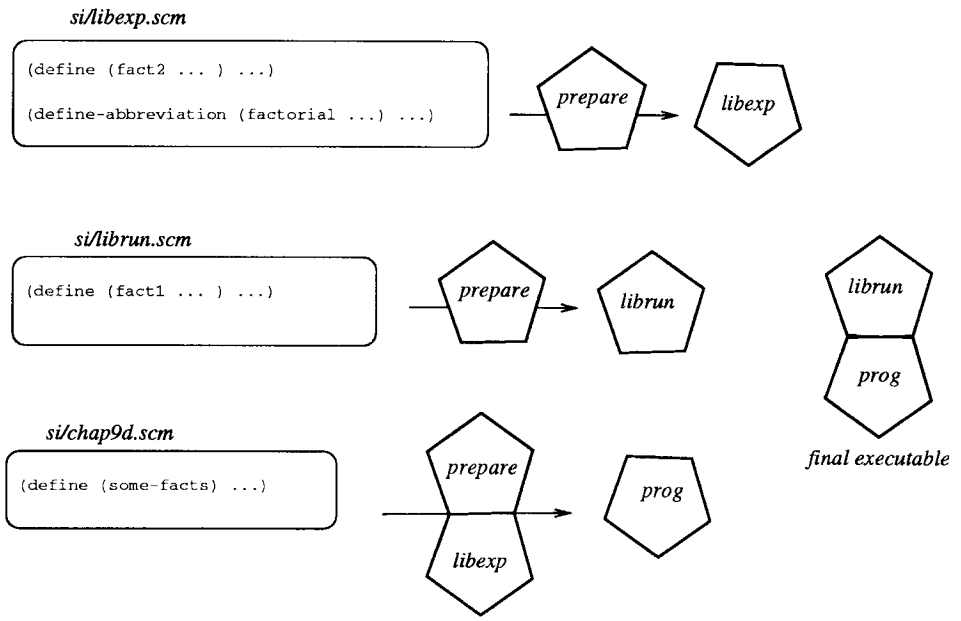


Figure 9.3 Multiple worlds: building an application

state of the macro expander is thus included in the interaction loop. According to this hypothesis, it is logical for the evaluator connected to endogenous macro expansion (the one we've called `macro-eval`) to be simply the `eval` function of the evaluator (but with all the possible variations that we saw in Chapter 8). [**see p. 271**] Since the world is unique, macro expansion has both read- and write-access to everything present in this world. It might seem easy then not to distinguish between the expansion library and the execution library since it suffices to submit this to the interaction loop:

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
(define-abbreviation (factorial n)
  (if (and (integer? n) (> n 0)) (fact n) '(fact ,n)) )
```

The problem is that now we make sharing (or selling) software very difficult because it is hard and often even impossible to extract just what's needed to execute or regenerate the program that we want to share. To be sure that we've forgotten nothing, we can deliver an entire memory image, but doing so is costly. We might try a *tree-shaker* to sift out anything that's useless. (A tree-shaker is a kind of inefficient garbage collector.) Actually, as you can see in many Lisp and Scheme programs available on Internet, we should abstain from delivering programs containing macros: either we deliver everything expanded (as *macroless* programs), or we deliver only programs with local macros. And at that point, we've negated all the expressive power that macros offered us!

If you don't believe in tools for automatic extraction, there's nothing left but to do it ourselves. To do so, we must distinguish three cases: the resources useful

only for macro expansion, those useful only for execution, and finally (the most problematic) those that are useful in both phases.

In order to prepare programs asynchronously, there's the function **compile-file**. Two variations oppose each other in the unique world. One might be called the *uniquely unique world.* The difference between the two concerns the macros available to prepare the module submitted to **compile-file**. Here are the three solutions that we find in nature.

1. In the uniquely unique world, there is only one macro space, and it is valid everywhere. Current macros are thus those that can use the expressions of the module being prepared. That's the case in Figure 9.4 where the macro **factorial** defined in the interaction loop is quite visible to the compiled module. Here's a subquestion then: what happens if the module being prepared defines macros itself?

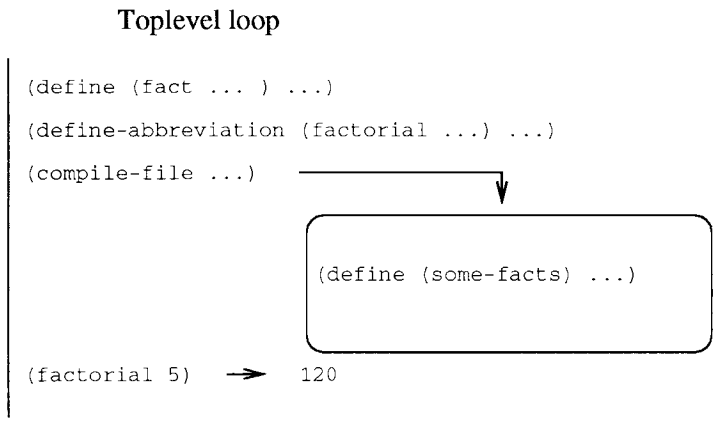


Figure 9.4 Uniquely unique world

   (a) In the uniquely unique world, these new macros are added to the current macros so the macro space continues to grow. (At least it does so unless we suppress macros altogether.) That's the case in Figure 9.5 if the expression **(factorial 5)** submitted to the interaction loop returns 120.

   (b) Another response would be that macros defined in the module are visible only to the preparation of that module. Thus we distinguish *super global macros* from macros *only global* to the modules. In Figure 9.5, that would be the case if the expression **(factorial 5)** were submitted to the interaction loop and yielded an error.

2. Finally, if the world is not uniquely unique, then the only macros visible to the module are those that the module defines endogenously. In that case, there is a space for macros for the interaction loop, and there are initially empty spaces for macros specific to each prepared module. However, even

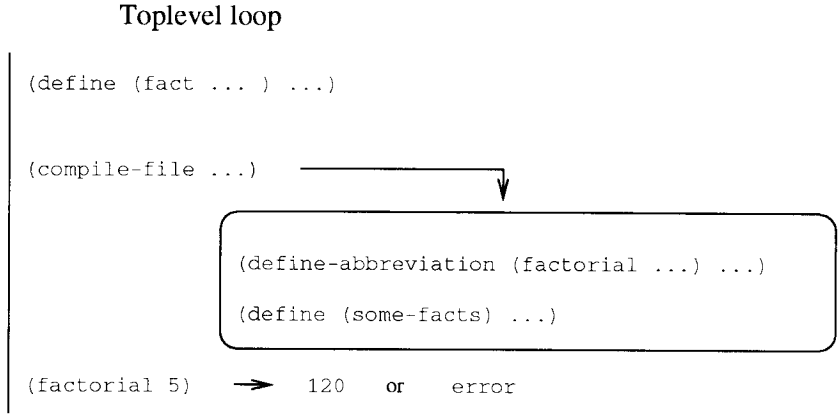


Figure 9.5 Unique world with endogenous compilation

though these macro spaces are separate, macro expansion still takes place in a unique world and thus can take advantage of all the global resources. For example, see Figure 9.5, where the macro `factorial` local to the compiled module can use the function `fact` from the interaction loop.

By looking closely at Figures 9.4 and 9.5, you see that in every case, the function `fact` belonging to the expansion library of `factorial` has been defined at the level of the interaction loop. This being so, it is globally visible even though it is probably useful only to the macro `factorial`. We could put it inside the macro, making it an internal function, as in `si/chap9b.scm`). [see p. 323]

What would happen if `fact` were useful to more than one macro? The problem is that the macro expander knows how to recognize only definitions of macros although we would often like to define global variables, utility functions, or various other data for the exclusive use of a set of several macros. Even though the hypothesis of multiple worlds in exogenous mode naturally insures that, that's not the case for endogenous mode nor for the unique world. Moreover, the principles of this hypothesis introduce the possibility of forcing an evaluation inside the macro expander. In COMMON LISP and other versions of Scheme, that practice is known as `eval-when`. To avoid interference, we'll name the practice `eval-in-abbreviation-world` for ourselves.

The purpose of the form `eval-in-abbreviation-world` is to make it possible to define constants, to define utility functions, to carry on any sort of computation inside and for the sole use of the macro expander. In fact, that form explicitly recognizes the need to separate that world from the exterior world of evaluation. In Figure 9.6, the function `fact` is defined in the evaluator of the macro expander, and `factorial` can thus use it to expand `some-facts`. On returning to the interaction loop, and depending on whether the worlds have "leaks" or not, `factorial` and/or `fact` can be visible and invokable again.

To program that behavior, all we have to do is ask the macro expander to recognize forms beginning with the keyword `eval-in-abbreviation-world` and

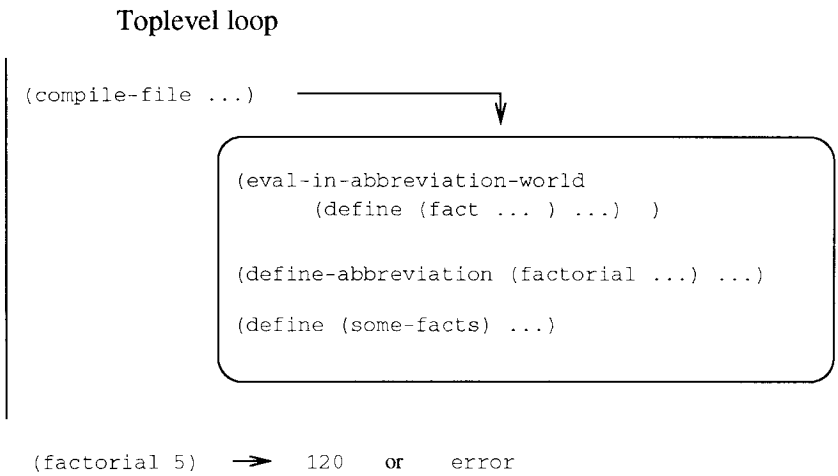


Figure 9.6 Endogenous evaluation

to evaluate them on the fly with the appropriate evaluator, that is, the one that we called **macro-eval** in the function **naive-endogeneous-macroexpander**.

```
(define (expand-expression exp macroenv)
  (define (evaluation? exp)
    (and (pair? exp)
         (eq? (car exp) 'eval-in-abbreviation-world) ) )
  (define (macro-call? exp)
    (and (pair? exp) (find-expander (car exp) macroenv)) )
  (define (expand exp)
    (cond ((evaluation? exp) (macro-eval `(begin . ,(cdr exp))))
          ((macro-call? exp) (expand-macro-call exp macroenv))
          ((pair? exp)
           (let ((newcar (expand (car exp))))
             (cons newcar (expand (cdr exp))) ) )
          (else exp) ) )
  (expand exp) )
```

The function **macro-eval** is the evaluator hidden behind macro expansion. It can be seen as an instance of the usual **eval** function, but provided with a clean global environment, not shared with the evaluator that the interaction loop uses. In that case, and for Figure 9.6, **(factorial 5)** and **(fact 5)** would both be in error.

Forms appearing in **eval-in-abbreviation-world** must be considered as executable directives included in the S-expression that is being expanded. In particular, these forms can use only what will later be the current lexical environment. We cannot write this:

```
(let ((x 33))                                          WRONG
  (eval-in-abbreviation-world (display x)) )
```

Again, if we want to introduce a variable locally for the sole benefit of the macro

expander—something that **eval-in-abbreviation-world** does not know how to do since it is fundamentally just an **eval** at *toplevel*—we can exploit **compiler-let** in COMMON LISP I [Ste84] but not in COMMON LISP II [Ste90]. All we have to do is make the function **expand-expression** recognize it.

Expansion cannot reach the final execution environment. Reciprocally, the final execution environment cannot reach the expansion environment. Fortunately, then, we cannot write this either:

```
(define-abbreviation (foo ...) ... )
(apply foo ... )                                    WRONG
```

Once we have **eval-in-abbreviation-world**, it's simple to explain **define-abbreviation** again: it's just a macro itself.

```
(define-abbreviation (define-abbreviation call . body)
  `(eval-in-abbreviation-world
     (install-macro! ',(car call) (lambda ,(cdr call) . ,body))
     #t ) )
```

Now the main point is that we can place definitions of macros everywhere and not just in toplevel position. Accordingly, if we want to organize several expressions into one unique sequence, we can write this:

```
(begin (define-abbreviation (foo x) ... )
       (bar (foo 34)) )
```

However, as a question of good taste, that should not be based on a too precise expansion order, nor should it lead to such slack writing as this:

```
(if (bar) (begin (define-abbreviation (foo x) ...)          BAD TASTE
                 (hux) )
          (foo 35) )
```

Most macro expanders distinguish macros at toplevel from others. For example, that's what distinguishes internal from global instances of **define**. Macro expanders also insure that toplevel expressions are treated sequentially from left to right. For that reason, we can define a class and then its subclasses in order.

In fact, the preceding examples have hidden a few implementation details due to the fact that several evaluators co-exist so not all the preceding definitions aim at the same evaluator. In the same way as with reflective interpreters, there's a problem of accessing the data structure that defines the current macros. [see p. 271] The form **install-macro!** in the definition of **define-abbreviation** will be evaluated by **macro-eval**, but it must modify the set of current macros as scanned by the function **find-expander**. However, **find-expression** is not evaluated by **macro-eval** but by **eval**! Well, these problems are not important for the moment.

One aim of **eval-in-abbreviation-world** is to introduce definitions of global variables in the macro expander because often macros behave in a way that depends on the underlying implementation. For example, a macro that constructs a call to the function **apply** in its macro expansion can inspect its environment to know whether the function **apply** found there is binary (which is sufficient for R^{2}RS or n-ary (as in R^{4}RS). However, checking its own environment means that the macro being expanded has the same environment as its target. In the case of complicated software (and we'll look into this case in MEROON later), it is sometimes useful to expand something for another implementation, in which case it is necessary for the

target implementation to be defined by *features* that can be inspected. You might see macros like this one:

```
(define-abbreviation (apply-foo x y z)
  (if (memq 'binary-apply *features*)
      `(apply foo (cons x (cons y z)))
      `(apply foo x y z) ) )
```

The variable ***features*** must be visible from the macro expander. That fact means that it must have been defined beforehand by means of **eval-in-abbreviation-world**. We can thus characterize an implementation by defining the variable ***features*** to be able to consult the macros under consideration. In the expression that follows—even though its appearance is tricky—the form **define** is at toplevel with respect to the macro language, so it is a definition of the global variable ***features***.

```
(eval-in-abbreviation-world
  (define *features* '(31bit-fixnum binary-apply)) )
```

9.6.3 Simultaneous Evaluation

An important variation often present in any hypothesis about the unique world is evaluation occurring simultaneously with preparation. As expressions are gradually expanded, they are evaluated by the interaction loop. That's what the following definition suggests:

```
(define (simultaneous-eval-macroexpander exps)
  (define (macro-definition? exp)
    (and (pair? exp)
         (eq? (car exp) 'define-abbreviation) ) )
  (if (pair? exps)
      (if (macro-definition? (car exps))
          (let* ((def       (car exps))
                 (name      (car (cadr def)))
                 (variables (cdr (cadr def)))
                 (body      (cddr def)) )
            (install-macro!
             name (macro-eval `(lambda ,variables . ,body)) )
            (simultaneous-eval-macroexpander (cdr exps)) )
          (let ((e (expand-expression (car exps) *macros*)))
            (eval e)
            (cons e (simultaneous-eval-macroexpander (cdr exps))) ) )
      '() ) )
```

Notice that two evaluators appear in that definition: **eval**, the evaluator in the interaction loop; and **macro-eval**, the evaluator in the macro expander. We have distinguished them from each other because they represent distinct processes occurring at distinct times.

9.6.4 Redefining Macros

The function **install-macro!** implied that macros could be redefined: if we redefine a macro, we modify the state of the macro expander thus altering the treat-

ments that remain for it to do. When we are using an interpreter and debugging a macro, it's fine that we can redefine that macro and test it through some function that we define to contain a site where this macro is called. That means that the interpreter delays the expansion of macros as long as possible. However, most of the time, expressions are expanded only once, and certainly to insure repeatable experiments, expressions should be expanded only once. As a consequence, those expressions become independent of any possible redefinitions of macros. In that sense, macros behave in a way that we could call hyperstatic. [see p. 55]

9.6.5 Comparisons

One of the main problems with macros is that it is difficult to change them if we're not satisfied with the system that an implementation provides. In fact, that difficulty has probably vetoed the unbridled experimentation that other parts of Lisp and Scheme have taken advantage of.

The system of multiple worlds in exogenous mode seems more precise but less widely distributed. The system of multiple worlds in endogenous mode is only an instance of the exogenous mode with an expansion engine that predefines certain macros like `define-abbreviation` and others. Finally, the unique world is the most widespread, but the least easy to master in terms of delivering software. This comparison among them is a little sketchy simply because it does not take into account two new facets that we'll discuss now: compiling macros and using macros to define other macros.

Compiling Macros

In what we've covered up to now, the macro expansion of a call to a macro is most often the work of an interpreter. For that reason, its efficiency seems compromised especially if the transformation of the programs produced by the macro is long and complicated (though that is rare). In the multiple world in exogenous mode [see p. 315], we build an *ad hoc* compiler by assembling prepared (that is, compiled) modules so we get efficiency. In endogenous mode, one solution is to use a compiling function, `macro-eval`. In the unique world, we can use `compile-file` and `load`. The problem is how to compile a definition of a macro?

The question is not trivial! The macro expanders that we've presented checked the expressions to expand and did so in order to find the forms `define-abbreviation` and to define those same macros on the fly. We must thus distinguish compilation of the macro from its installation. If `define-abbreviation` is a macro that expands into the form `install-macro!`, then we're home free. In contrast, if `define-abbreviation` is a syntactic keyword, then we merely have to insure that the body of a macro is only an interpreted call to a compiled function. Once it has been compiled, we must again load the prepared program, and at that point, several problems arise: what, for example, does `(eval-in-abbreviation-world (load file))` mean? The function `load` is just a wrapper around `eval`, but in the macro expansion world, we must use `macro-eval`, not `eval`! We won't even talk about an explicit call to `eval`, as in `(eval-in-abbreviation-world (eval '(define-abbreviation ... )))`, where, once again, `eval` must refer to

`macro-eval`, rather than the evaluator in the interaction loop. This (non-)discussion clearly shows that the two global environments belonging to `eval` and `macro-eval` really are separate and provide very different entities under the same name.

We come back to the point that the macro definer has to be a macro itself; it cannot be just a syntactic marker that the macro expander searches for. Accordingly, we separate the preparation of a macro from its installation. We'll take up this point again later. [see p. 336]

Macros Defining Macros

From time to time, it's useful to define macros that generate other macros themselves. [see p. 339] This is not just some weird habit but a logical application of the principles of abstraction and freedom of expression. For example, in an object system, we might want to define the class `Point` where the accessors `Point-x` and `Point-y` should be macros instead of functions. [see p. 424] In that case, the macro `define-class` must generate those new macros.

Let's take the example of another macro, `define-alias`, which defines its first argument to be equal to its second. We'll give you two variations of it, one of them enclosing *backquotes* in *backquotes*.

```
(define-abbreviation (define-alias newname oldname)
  `(define-abbreviation (,newname . parameters)
     `(,',oldname . ,parameters) ) )
(define-abbreviation (define-alias newname oldname)
  `(define-abbreviation (,newname . parameters)
     (cons ',oldname parameters) ) )
```

Conventionally, where the result of a macro expansion is expanded again, if `define-abbreviation` is a macro, then there is no problem. In contrast, if `define-abbreviation` is merely a syntactic marker, then it's quite likely that the macro expander will not perceive this definition if it does not analyze the result of macro expansions. Consequently, we'll abandon the idea of a syntactic marker in favor of predefined and even primitives macros because we wouldn't know how to create them if they were missing.

9.7 Scope of Macros

The scope of local macros, introduced by `let-syntax` or `letrec-syntax` in Scheme, poses no problems. That's not the case, however, for macros defined by `define-syntax` because we can distinguish several cases for using macros. In this section, we'll take MEROON for discussion. It is an object system built on top of Scheme; it has already been ported to many implementations of Scheme, both interpreted and compiled. The essence of the system MEROON is in MEROONET. (See Chapter 11.) Three kinds of macros exist in that system:

1. **Type 1—occasional macros**: An occasional macro is defined in one place and used immediately afterwards. We can thus transform it into a local macro defined by a form, `let-syntax` or `macrolet`, if we have that available,

though of course that's not the case everywhere. For example, the function **make-fix-maker** in MEROONET builds a "triangle" of closures (**lambda (a b c ...) (vector cn a b c ...)**), but it does so "by hand." [see p. 434] The following macro does that automatically.

```
(define-abbreviation (generate-vector-of-fix-makers n)
  (let* ((numbers (iota 0 n))
         (variables (map (lambda (i) (gensym)) numbers)) )
    `(case size
       ,@(map (lambda (i)
                (let ((vars (list-tail variables (- n i))))
                  `((,i) (lambda ,vars (vector cn . ,vars))) ) )
              numbers )
       (else #f) ) ) )
```

An occasional macro can be abandonned, once it's been used. Its scope is greatly restricted: it does not go beyond the module where it appears.

2. **Type 2—macros local to a system**: The definition of MEROON is organized into about 25 small files. A number of abbreviations appear throughout those files, for example, the macro **when**. The scope of that macro is thus the set of files making up the source of MEROON, but no more than that, because we have no right to pollute the world exterior to MEROON.
3. **Type 3—exported macros**: MEROON exports three macros: **define-class**, **define-generic**, and **define-method**. These macros are for MEROON users but they also bootstrap MEROON. The macro **when** cannot appear in the expansion of macros of type 3 because allowing that would confer a greater scope on **when** than we planned for. The language of the macro expansion of exported macros is the user's language, not the language of MEROON itself.

Not only are there three kinds of macros; there are also three ways of using them, as illustrated in Figure 9.7. Those three ways are:

1. **To prepare MEROON sources**: This use involves expanding the source of MEROON and thus getting rid of all uses of type 1, 2, and 3. In contrast, something from the definition of macros of type 3 necessarily has to live somewhere since they are exported.
2. **To prepare a module that uses MEROON**: Preparing a module that uses MEROON means expanding MEROON macros of type 3 that this module uses.

   Consequently, in the thing that prepares this module, we have to install type 3 MEROON macros. In other words, we have to graft the expansion library of type 3 macros onto the preparer. Notice, though, that we don't actually need the expansion library of MEROON type 1 or 2.
3. **To generate an interpreter providing the user with MEROON**: In this case, we want to construct an interaction loop that provides type 3 MEROON macros. Consequently, those macros have to be installed, along with their expansion library in the macro expander connected to the interaction loop.

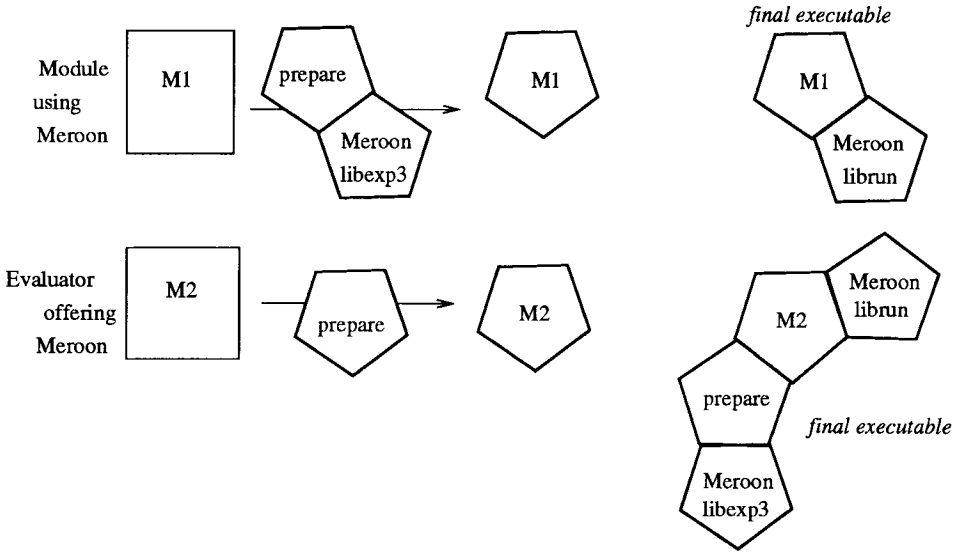


Figure 9.7 Usage of macros

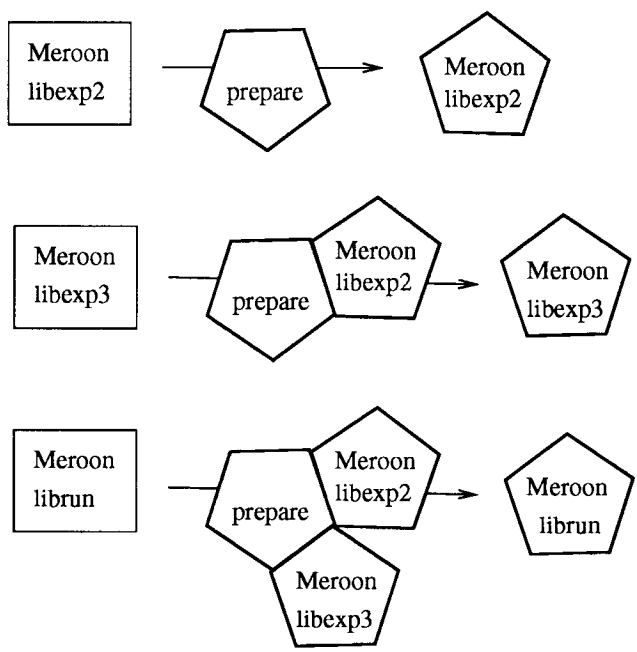


Figure 9.8 Bootstrapping MEROON

Figure 9.8 shows one way of getting MEROON modules by bootstrapping. The expansion library of type 2 macros are only for preparing the rest of the MEROON source. You see that the construction of a complicated piece of software resembles a T-diagram, as in [ES70], where we handle various languages and their implementations.

When you consider the types of macros and their various uses that we've just explained, you observe that multiple worlds are well adapted for this, whereas the unique world generally does not support limiting the scope of a macro to the place where it is used. The idea of a *package* would let us organize the names we use more precisely (as in COMMON LISP or ILOG TALK) where all we have to do is give macros names that are restricted to internal use so we avoid any future collisions. In contrast, in a unique world, those three different uses of macros that we carefully distinguished are all mixed up. To use MEROON, you just load it and that's all!

Chapter 11 defines MEROONET. Even though it provides three macros, it does not use them for its own purposes (to avoid unpleasant questions). It defines macros (type 3, according to our terminology) with the macro **define-meroonet-macro**; we'll get into the details of its implementation later. Our first task, when confronted with a system of predefined macros, is to determine what type it is. Then we must set it in motion, and (since we generally cannot access functions like **install-macro!**) we have to do everything with the only macro definer in the implementation. By "everything," we mean from the simple equivalence between **define-meroonet-macro** and **define-abbreviation** up to the following convoluted definition that insures that the macro is simultaneously useful right away and will eventually be useful in the prepared file where it appears when that file is loaded dynamically.

```
(define-abbreviation (define-meroonet-macro call . body)
  `(begin (define-abbreviation ,call . ,body)
          (eval '(define-abbreviation ,call . ,body)) ) )
```

9.8 Evaluation and Expansion

Macro expansion and evaluation are intimately connected in the interaction loop since they are immediately linked to each other. This section looks more closely at that couple.

Macro expansion is the first phase of preparation. Evaluation is the phase that follows preparation. The function **eval** represents only evaluation, while the language that **eval** accepts is merely the pure language, stripped of all macros. If we want to take advantage of macros, then, we ourselves must expand the expressions that we provide to **eval**. Accordingly, in a multi-windowed evaluation system, we could see two different macro expanders working simultaneously in different windows. However, that is not really a practical arrangement; it doesn't really conform to current practice, and moreover, we would probably lose the usual macros, like **cond**, **let**, or **case**, in doing that. For all those reasons, the evaluation function **eval** is usually provided as **(lambda (e) (pure-eval (macroexpand e *macros*)))**, where **pure-eval** is the evaluator for the pure language while **macro-expand** is the public function for macro expansion and ***macros*** contains

the macros known to the interaction loop.

What we've just said about **eval** is also true about **macro-eval** which has its own ***macros*** variable containing only the macros known by the preparation. A great many questions arise here about this pair, macro expansion and evaluation. We've already mentioned [see p. 277] the existence of **eval/ce** for *evaluate in the current environment*. That special form captured the entire visible lexical context. Since **eval/ce** knows how to expand the expressions it receives, does it capture the macro expansion context? In other words, if we write this:

```
(let-syntax ((foo ...))
  (eval/ce (read)) )
```

then will an expression read and containing a call to **foo** be expanded correctly by **eval/ce**? If yes, then not only must **eval/ce** keep a trace of the lexical context, but it must also save the memory state of the entire macro expander in order to keep its ability to expand **foo**. Since that seriously undermines the independence of macro expansion, we regard that practice as in seriously bad taste. However, the right solution is that evaluation should be pure and not force macro expansion right away, but then that's not practical, so etc. etc.

When we define a macro, the macro is visible for the rest of the macro expansion. Let's assume that we've defined the useful macro **when**. Then can we write this?

```
(define-abbreviation (whenever condition . corps)
  (when condition (display '(whenever is called)))
  `(when ,condition . ,corps) )
```

That macro uses the word **when** twice. The second occurrence is located in the expanded macro so it poses no problem. In contrast, the first one appears in the computation of the macro expansion and is thus evaluated by the evaluator for macro expansion, which *a priori* does not know about **when**: the macro language is not the one that we are expanding! For the macro **when** to be available for use by the definition language for macros, we should have already provided the following definition:

```
(eval-in-abbreviation-world
  (define-abbreviation (when condition . body)
    `(if ,condition (begin . ,body)) ) )
```

The problem is similar in this expression:

```
(define-abbreviation (foo)
  (define-abbreviation (bar)
    (when ...)
    (wrek) )
  (hux) )
```

The internal definition of **bar** is *a priori* destined to enrich the macro language, not the current language. The preceding expression defines the macro **bar** as the macro possible to use to write the macro. Even if the semantics seems clean, the pragmatics are a little off because we now have a problem of infinite regression. The language for defining macros and the language for defining the definition language for macros are not necessarily the same. In other words, as in the preceding example, we can't be sure that the form **when** present in the definition of **bar** is included and understood. If we had truly different languages, the problem would

be even more apparent. For example, you can imagine that macros are handled by `cpp` and that for `cpp` macros themselves come from a `perl` program. Then it becomes obvious that the definition of a macro on one level has implications only on that level.

How do we resolve such a problem? One possibility is to stick to the semantics and have multiple levels of language, but (fortunately!) we rarely need to get higher than the second level. There we would again find the problems of levels of language that we saw with reflective interpreters. [see p. 302]

Another possibility would be to fuse all language levels used by the macro expander. This boils down to adopting the hypothesis of the unique world for the evaluator in macro expansion. The circle closes in on us again, and everything that we carefully separated is all mixed up once more.

A third possibility—the one adopted by R^{4}RS—is to restrain the language to express macros to only filtering and reconstructing capabilities. Macros can no longer be defined in that language!

In short, once again it's very important to distinguish the languages that come into play. In an implementation of Scheme that supports distribution and parallelism, it's probably not a good idea to make the expansion of macros distributed and parallel as well. We might even forbid the writing of macros that use non-local continuations, like this:

BAD TASTE

```
(define-abbreviation (foo x)
  (call/cc (lambda (k)
             (set! the-k k)
             x )) )
(define-abbreviation (bar y)
  (the-k y) )
```

But how would we enforce that rule?

9.9 Using Macros

This section goes into the details of how macros are typically used. While we agree that their purpose is to transform programs, we might have many different reasons for transforming programs. Among them, we distinguish these:

- *Shortcuts* limit the number of characters a user must type (especially at toplevel or during debugging). If we want to write `(trace foo)` to make visible all the calls to the function `foo`, then we need a macro that does not evaluate `foo`.
- *Beautification* lets us mask disharmonious syntax. For example, we might use `bind-exit` rather than `call/ep` to avoid an extra `lambda`.
- *Masks* let us hide the underlying implementation—a primordial goal. We need to hide the implementation details when they are likely to change or when it's not a good idea to make them accessible for general use. As examples, consider `define-class` in MEROONET or `syntax-rules` in Scheme.

Among the masks, we also find macros whose aim is to abstract porting problems. Using a macro, like `apply-foo` [see p. 331] to hide the fact that there are

two kinds of **apply**, a binary and an n-ary, is a current practice. Another porting problem crops up with the Boolean value of () in Scheme, which is True in R^{4}RS but not necessarily so in R^{3}RS. We can get around this problem by using a macro everywhere; we'll call it **meroon-if** and define it like this:

```
(define-abbreviation (meroon-if condition consequent . alternant)
  `(if (let ((tmp ,condition))
         (or tmp (null? tmp)) )
       ,consequent . ,alternant ) )
```

Of course, we can avoid that problem if our program is robust, but the fact is, some very interesting software has been written in a style that is not entirely robust, alas. Prefixing it with such a macro is one way of porting it toward another implementation.

Among the masks, we also find macros whose role is to insure that certain optimizations will be carried out independently of the target compiler. One important improvement—known as *inlining*—is to integrate functions. Some compilers offer such a directive, but inasmuch as that directive is not uniformly distributed, and its application is problematic beyond the frontiers of a given module, the simplest approach is to handle this integration ourselves. To make the calls to a function "inlinable," we associate the function with a macro of the same name. We would thus define something like this:

```
(define-abbreviation (define-inline call . body)
  (let ((name      (car call))
        (variables (cdr call)) )
    `(begin
       (define-abbreviation (,name . arguments)
         (cons (cons 'lambda (cons ',variables ',body))
               arguments ) )
       (define ,call (,name . ,variables)) ) ) )
```

That macro suffers from portability problems itself because in some dialects, it is not possible to have a macro and a function of the same name simultaneously. In other dialects, that practice is allowed, and the second definition (the function) will completely replace the first (the macro) and in consequence, the macro will no longer be accessible. Anyway, it is nearly impossible for the function to be recursive without making the macro loop (though see [Bak92b] for more detail). Finally, when a function is inlined, it's best to avoid using it as a first class value (for example, as the first argument to **apply**) since that allows you not to define the associated function.

The preceding macro is easily inlined in multiple worlds since the definition of the function is merely expanded and in fact, expanded by means of the macro that precedes it. In the resulting expanded code, only the function appears in all its uses, while all the places where the macro of the same name is called will have been expanded—just the effect we were counting on.

9.9.1 Other Characteristics

The macros in Scheme R^{4}RS have four outstanding characteristics:

1. they are hygienic; (we'll get to that idea in Section 9.10);

2. they are defined by *filtering*;
3. they build the expanded macro by substitution;
4. they have no internal state.

The advantage of defining macros by filtering is that we can then very finely control the form that the call to the macro has to respect. Moreover, in writing the definition, we do not have to write the code for checking that conformity. For example, most macros where the last argument forms an implicit `begin` don't test whether the ultimate `cdr` is really `()`. That's automatically verified by a filter, and an error message is sent if need be. There are efficient compilers for filters, well described in the literature, such as [Que90b, QG92, WC94].

Construction by substitution is part of *backquoting* notation but it restricts computations to only those operations that can be carried out on lists. In particular, it does not support arithmetic operations. **[see Ex. 9.2]**

The fact that macros have no internal state is a little more inconvenient. Because of this lack, we cannot have contextual macros, like `define-class`, which should be able to keep the hierarchy of known classes updated so that we could find necessary information there when subclasses are introduced. We would like to know the names and number of inherited fields, for example.

Even so, you can imagine a macro, like `date`, that returns a character string indicating the current date at the time it is expanded. That macro could work like Walter Tichy's RCS or Eric Allman's SCCS to maintain versions of software.

9.9.2 Code Walking

Most macros that we write take one expression and return another that they build from the subterms of the first expression. Those are rather superficial macros that don't need to frisk the input expression. However, a macro that translates an infix arithmetic expression into a Lisp expression in prefix notation, for example, needs to analyze its argument more deeply. Let's take the case, for example, of the macro `with-slots` from CLOS; we'll adapt it to a MEROONET context. The fields of an object—let's say the fields of an instance of `Point`—are handled by read- and write-functions like `Point-x` or `set-Point-y!`. It would be simpler to handle them directly by the name of their fields, `x` or `y`, for example, in the context of defining a method. Similar to Smalltalk in [GR83], we could thus write this:

```
(define-handy-method (double (o Point))
  (set! x (* 2 x))
  (set! y (* 2 y))
  o )
```

in place of this:

```
(define-method (double (o Point))
  (set-Point-x! o (* 2 (Point-x o)))
  (set-Point-y! o (* 2 (Point-y o)))
  o )
```

The new macro, `define-handy-method`, thus must now check its own body to convert the references and assignments there to variables with the name of fields of the class being discriminated upon. To access these fields, we can use accessors

that do not test whether their argument belongs to the class, since the fact that the argument belongs to the class is guaranteed by discrimination. In consequence, access is simpler and increasingly more efficient.

To produce these effects, we must first recall that the body of the method is not a program but rather an S-expression before macro expansion before analysis. For that reason, we must have access to the mechanism for macro expansion. Since macro expansion can give rise to local macros, we must also have access to the expander itself, the one that takes care of the original form. Without getting tangled up in the details, we'll assume that the expander is the value of the variable (whether local or global) named `macroexpand` and that it is always visible from the body of macros.

Once the body of `define-handy-method` has been expanded, then it's a real program, and at that point it has to be analyzed. Well, we've been analyzing programs since the beginning of this book; we do it by recognizing the special forms of the language that is being analyzed. We do not go off searching for references to fields in quotations; however, we must take into account binding forms that can hide fields. It's not too complicated a task to write a code walker, as in [Cur89, Wat93] **[see p. 340]**; what's hard is to recognize precisely the set of special forms in the language—or rather the implementation—that we're using.

Special forms are points of reference in an implementation, but an implementation sometimes has private features, such as:

- supplementary special forms (as with `let` or `letrec`) or hidden special forms belonging to the implementation (as with `define-class`);
- special forms implemented as macros (for example, with `begin`).

In the absence of reflective information about the language, it is difficult to get out of this dilemna because we have to ask, "How do we handle `begin` forms if they've disappeared? How do we find conditionals when `if` is a macro that expands into the special form `typecase`? How do we inspect an unknown special form like `define-class` when it involves substructures (such as definitions of fields) that systematically resemble functional applications? How do we guess that `bind-exit` is a special form establishing a binding that can hide a variable of the same name?"

A good compromise would be to freeze the set of special forms and forbid any more or fewer of them. Doing that means that the implementations that use special macros would not be able to do so in any phase later than macro expansion. However, lacking reflective information about the language and its implementations, we cannot write a portable code walker in Scheme, so we have to give up writing `define-handy-method`.

9.10 Unexpected Captures

In recent years, the idea of *hygiene* with respect to macros has been carefully studied in [KFFD86, BR88, CR91a]. An expanded macro contains symbols that are in some respects "free"; that is, they are unattached and thus susceptible to interference, either capturing or being captured with the context where the macro is used. Here's an example with both effects:

```
(define-abbreviation (acons key value alist)
  '(let ((f cons)) (f (f ,key ,value) ,alist)) )
(let ((cons list)
      (f    #f) )
  (acons 'false f '()) )
```

The reference to **cons** appearing in the expanded macro **acons** referring to our familiar **cons** is going to be captured by the local variable **cons** present in the **let** form surrounding the place where **acons** is called. Conversely, the expanded macro establishes a binding for the variable **f** which will prevent the second argument of the call to **acons** from taking the value **#f** would normally have taken. In short, the macro has intercepted the variable **f**. What a mess!

When we make macros *hygienic*, we automatically escape from such problems. Let's look at how those problems are conventionally handled.

For more than thirty years, users have been solving that second problem in Lisp simply by renaming. Specifically, the variable **f** which the expanded macro introduces must not interfere at all with the lexical environment where the macro is used. For that reason, we'll carry out an α-conversion and generate a new and inimitable **gensym** in order to protect that variable. Thus, quite serenely we'll write this:

```
(define-abbreviation (acons key value alist)
  (let ((f (gensym)))
    '(let ((,f cons)) (,f (,f ,key ,value) ,alist)) ) )
```

The first problem we mentioned—producing a free reference to **cons** in the expanded macro—is more complicated to handle. In the case of the present macro, what we want is for the writer of the macro to be able to specify that the reference to **cons** is the reference to the global variable **cons** and nothing else. In fact, the global variable **cons** was the one visible from the definition of the macro **acons**. From that observation we derive the first rule of macro hygiene: the free references in the expanded macro are those that were visible from the place where the macro was defined.

In the case of **cons** and in certain Lisp or Scheme systems, there is often a method to reference a global variable independently of the context; the method is equivalent to a form like **(global cons)** or **lisp:cons**. However, the following example will show you that we may also want to associate a local binding with a variable of an expanded macro.

```
(let ((results '())
      (compose cons) )
  (let-syntax ((push (syntax-rules ()
                       ((push e) (set! results (compose e results))) )))
    π
    results ) )
```

In the entire body π of this expression, we want the local macro **push** to compose its argument with the contents of the variable **results** and assign **results** with that result. To be hygienic, we must insure that no other internal binding to π can modify the sense of **push**. Basically, then, there are two solutions:

1. rename all the internal variables in π so that **results** and **compose** are always visible and unambiguous there;

2. rename **results** and **compose** in a consistent way; that is, in **let** bindings, in the definition of **push**, and of course in the last expression of the body of **let**.

Even though we've talked about the "capture" of bindings, there is no such thing in hygienic macros. They don't capture bindings because bindings don't yet exist in hygienic macros since there we're still in the macro expansion phase! Even more strongly, R^{4}RS does not reserve keywords so it's possible to define a macro with the name of a special form. As a consequence, even the word **set!** apparently free in the expanded macro of **push** could be captured by a macro local to π. Thus we don't capture bindings, we capture meanings!

Macro hygiene is a remarkable and attractive property, but we can't adopt it whole-heartedly because there are important macros that are not in fact hygienic. The most popular example is the macro **loop**. We generally get out of it by means of the function **exit**. If we were to write this:

WRONG

```
(define-syntax loop
  (syntax-rules ()     ;should be (exit) instead
    ((loop e1 e2 ...)
     (call/cc (lambda (exit)
                (let loop () e1 e2 ... (loop)) )) ) ) )
```

then we would never get out of the loop because the variable **exit** introduced in the expanded macro cannot be captured (because of hygiene!) by a reference to **exit** in any of the forms **e1**, **e2**, ... If **exit** appeared among those expressions, its intention would be to mention what **exit** indicates at its calling site. The solution is to mention that the symbol **exit** must remain free in the expanded result. To do that, we must put it into the parentheses that follow the word **syntax-rules**. By default, it's good for everything to be hygienic. This is precisely what we wanted for the **loop** variable: to be free of any captures. However, from time to time, we have to break our own rule about hygiene.

Many competitive implementations of hygienic macros are available on the net, and they are well worth reading. Our solution will won't use filtering; it will not limit the kind of computations that can be carried out within macros; but it corresponds to a low-level implementation that we find simpler than those appearing in the literature, such as [CR91b, DHB93]. Our implementation offers a new variation where we explicitly mention the words whose meaning we want to freeze. Here's an example: the form **with-aliases** will take a set of pairs, each made up of a variable and a word, as its first argument; *for the duration of the macro expansion*, it will bind these variables to the meaning that those words have in the current context. The preserved meaning of those words is accessible from those variables, and it can be inserted in the expanded macro. The following parameters make up the body of the form **with-aliases**; their role is to compute the expanded result. Thus we would rewrite the preceding example like this:

```
(let ((results '())
      (compose cons) )
  (with-aliases ((setq set!) (r results) (c compose))
    (let-abbreviation  ( ( (push e)
                           `(,setq ,r (,c ,e ,r)) ) )
      π
```

```
    results ) ) )
```

In contrast, the preceding `loop` macro would be defined like this:

```
(with-aliases ((cc call/cc)(lam lambda) (ll let))
  (define-abbreviation (loop . body)
    (let ((loop (gensym)))
      `(,cc (,lam (exit) (,ll ,loop () ,@body (,loop)))) ) ) )
```

In those two examples, all the terms that we want to freeze are mentioned in a `with-aliases` which captures their meaning. They are then inserted in the expanded results and are thus independent of the context where these macros are used.

Backquote notation is not really appropriate here because the proportion of variant elements is very high. This system of macros is rather low-level because it is not automatically hygienic; rather, it merely offers the tools for being hygienic. In the next section, we'll develop this system of macros further.

9.11 A Macro System

This section describes the implementation of a system of macros with the following functions:

- `define-abbreviation` to define a global macro;
- `let-abbreviation` to define a local macro;
- `eval-in-abbreviation-world` to evaluate something in the macro world;
- `with-aliases` to preserve a given meaning.

In order to show the difference between preparation and execution more clearly, we will expand programs and then transform them into objects on the fly. The resulting objects can be handled in either of two ways: they can be evaluated by a fast interpreter, as before in Chapter 6 **[see p. 183]**; or they can be compiled into C by the compiler in Chapter 10, **[see p. 359]**. Consequently, we expand them in only one pass, and the forms are frozen into objects as soon as they are expanded.

9.11.1 Objectification—Making Objects

Reification—which we've already used in another context—is not exactly the same thing as objectification. First of all, the program being compiled is converted into an object. During its conversion, its syntax will be checked and normalized so that syntax will not be a source of any other eventual errors. This transformation resembles rapid interpretation (the two have the same goal), but this time, we're working *ad hoc*. Instead of closures without arguments to contain all the necessary ingredients for their evaluation, this time, we will make objects that can be evaluated, thus showing how robust this transformation is; additionally, these objects can be handled more generally as well.

Here is the list of classes that we need for these objects:

```
(define-class Program Object ())
(define-class Reference Program (variable))
```

```
(define-class Local-Reference Reference ())
(define-class Global-Reference Reference ())
(define-class Predefined-Reference Reference ())
(define-class Global-Assignment Program (variable form))
(define-class Local-Assignment Program (reference form))
(define-class Function Program (variables body))
(define-class Alternative Program (condition consequent alternant))
(define-class Sequence Program (first last))
(define-class Constant Program (value) )
(define-class Application Program ())
(define-class Regular-Application Application (function arguments))
(define-class Predefined-Application Application (variable arguments))
(define-class Fix-Let Program (variables arguments body))
(define-class Arguments Program (first others))
(define-class No-Argument Program ())
(define-class Variable Object (name))
(define-class Global-Variable Variable ())
(define-class Predefined-Variable Variable (description))
(define-class Local-Variable Variable (mutable? dotted?))
```

In that list, you can see several points that we've already covered. For example, closed applications are treated specially. Calls to functions are identified. The class **Program** contains only elements that can be evaluated: references and assignments are there. In contrast, instances of variables representing bindings are not programs. They are instances of the class **Variable**. There are few idiosyncracies in these definitions of classes. Generally, they have as many fields as there are syntactic components. However, the class of local variables has two Booleans: one indicating whether or not the local variable is assigned, and another indicating how it is bound. In particular, **dotted?** variables receive special treatment before they receive a list of arguments.

We assume that **read** (or some other equivalent returning the S-expression that was read) reads the expression to compile. In fact, it would be smart to read an expression by means of **cons** with five fields to store in which file, which line, and which column the expression was read so that we could get excellent warning messages in case of errors. Moreover, symbols should have a supplementary field containing any possible associated macro; then searching for such a macro would be really fast. Augmenting symbols and dotted pairs with extra arguments during macro expansion is not too costly nor cumbersome since there won't be anything left around after expansion.

Thus we walk through this S-expression, as it's being read, to convert it into an object of the class **Program**. In passing, its macros are identified and expanded. The principal function, **objectify**, takes the lexical preparation environment as its second argument. Basically, given a form, it analyzes the term in function position and starts the appropriate treatment. Treatments are not tangled up with one another any more; rather, the right ones are located in the **handler** field of objects of the class **Magic-Keyword** (according to the terminology of [SS75]). The function **objectify** itself paves the way for handling macros; we'll explain it later.

```
(define-class Magic-Keyword Object (name handler))
(define (objectify e r)
  (if (atom? e)
      (cond ((Magic-Keyword? e) e)
            ((Program? e)       e)
            ((symbol? e)        (objectify-symbol e r))
            (else               (objectify-quotation e r)) )
      (let ((m (objectify (car e) r)))
        (if (Magic-Keyword? m)
            ((Magic-Keyword-handler m) e r)
            (objectify-application m (cdr e) r)) ) ) )
```

There's nothing left to explain except the various specialized subfunctions for conversion. Most of them simply build an object where the fields are initialized with the results of the recursive inspection of the subterms of the corresponding S-expression. That's certainly the case of **objectify-alternative** and **objectify-sequence**; **objectify-sequence** analyzes the initial forms in order to normalize it in binary sequences.

```
(define (objectify-quotation value r)
  (make-Constant value) )
(define (objectify-alternative ec et ef r)
  (make-Alternative (objectify ec r)
                    (objectify et r)
                    (objectify ef r) ) )
(define (objectify-sequence e* r)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (let ((a (objectify (car e*) r)))
            (make-Sequence a (objectify-sequence (cdr e*) r)) )
          (objectify (car e*) r) )
      (make-Constant 42) ) )
```

The application is a little more complex since it must analyze the initial expression and categorize it as a closed application, a call to a predefined function, or a normal application. It categorizes on the basis of its analysis of the object corresponding to the term in the function position. The only obscure point here is one we've already mentioned, [see **p. 200**]: whether there is a description letting us know the arity of predefined functions and how to use it to compile them better. Later, we'll cover the class **Functional-Description** when we look at how to put in the predefined environment.

```
(define (objectify-application ff e* r)
  (let ((ee* (convert2arguments (map (lambda (e) (objectify e r)) e*))) )
    (cond ((Function? ff)
           (process-closed-application ff ee*) )
          ((Predefined-Reference? ff)
           (let* ((fvf (Predefined-Reference-variable ff))
                  (desc (Predefined-Variable-description fvf)) )
             (if (Functional-Description? desc)
                 (if ((Functional-Description-comparator desc)
                      (length e*) (Functional-Description-arity desc) )
```

```
                    (make-Predefined-Application fvf ee*)
                    (objectify-error
                     "Incorrect predefined arity" ff e* ) )
                (make-Regular-Application ff ee*) ) ) )
          (else (make-Regular-Application ff ee*)) ) ) )
(define (process-closed-application f e*)
  (let ((v* (Function-variables f))
        (b  (Function-body f)) )
    (if (and (pair? v*) (Local-Variable-dotted? (car (last-pair v*))))
        (process-nary-closed-application f e*)
        (if (= (number-of e*) (length (Function-variables f)))
            (make-Fix-Let (Function-variables f) e* (Function-body f))
            (objectify-error "Incorrect regular arity" f e*) ) ) ) )
```

The list of arguments is translated into a unique object (an instance of the class **Arguments**); the list ends with an object from the class **No-Argument**. The number of arguments can be determined by the generic function **number-of**.

```
(define (convert2arguments e*)
  (if (pair? e*)
      (make-Arguments (car e*) (convert2arguments (cdr e*)))
      (make-No-Argument) ) )

(define-generic (number-of (o)))
(define-method (number-of (o Arguments))
  (+ 1 (number-of (Arguments-others o))) )
(define-method (number-of (o No-Argument)) 0)
```

As usual, inside closed applications, we will distinguish the particular rare and cumbersome case of applied n-ary functions. Supplementary arguments are organized into a list; thus we modify the associated dotted variable so that it becomes normal again. [see p. 197]

```
(define (process-nary-closed-application f e*)
   (let* ((v* (Function-variables f))
          (b  (Function-body f))
          (o (make-Fix-Let
              v*
              (let gather ((e* e*) (v* v*))
                (if (Local-Variable-dotted? (car v*))
                    (make-Arguments
                     (let pack ((e* e*))
                       (if (Arguments? e*)
                           (make-Predefined-Application
                            (find-variable? 'cons g.predef)
                            (make-Arguments
                             (Arguments-first e*)
                             (make-Arguments
                              (pack (Arguments-others e*))
                              (make-No-Argument) ) ) )
                           (make-Constant '()) ) )
                     (make-No-Argument) )
                    (if (Arguments? e*)
```

```
                    (make-Arguments (Arguments-first e*)
                                    (gather (Arguments-others e*)
                                            (cdr v*) ) )
                    (objectify-error
                     "Incorrect dotted arity" f e* ) ) ) )
            b )) )
      (set-Local-Variable-dotted?! (car (last-pair v*)) #f)
      o ) )
```

Then we analyze abstractions and transform them by means of **objectify-function**. It uses **objectify-variables-list** to handle the list of variables and thus enrich the lexical environment in which the body of the function will be treated.

```
(define (objectify-function names body r)
  (let* ((vars (objectify-variables-list names))
         (b    (objectify-sequence body (r-extend* r vars))) )
    (make-Function vars b) ) )
(define (objectify-variables-list names)
  (if (pair? names)
      (cons (make-Local-Variable (car names) #f #f)
            (objectify-variables-list (cdr names)) )
      (if (symbol? names)
          (list (make-Local-Variable names #f #t))
          '() ) ) )
```

Finally, **objectify-symbol** carefully handles variables. It searches for them in the unique, static, current, lexical environment: **r**. If the variable is not found there, the variable is added to the mutable global environment by the function **objectify-free-global-reference**. That is, we've adopted here a way of automatically defining new variables.

```
(define (objectify-symbol variable r)
  (let ((v (find-variable? variable r)))
    (cond ((Magic-Keyword? v)       v)
          ((Local-Variable? v)      (make-Local-Reference v))
          ((Global-Variable? v)     (make-Global-Reference v))
          ((Predefined-Variable? v) (make-Predefined-Reference v))
          (else (objectify-free-global-reference variable r)) ) ) )
(define (objectify-free-global-reference name r)
  (let ((v (make-Global-Variable name)))
    (insert-global! v r)
    (make-Global-Reference v) ) )
```

The environment **r** is more or less a list of local variables followed by global variables and then the predefined variables. The static environment does not contain the values of these variables, since the values result from subsequent evaluation. This environment is represented by a list of instances of **Environment**. It's possible to extend this environment by local variables or by new global variables. New global variables are appended physically to the global part of the environment; we know how to find it again with the function **find-global-environment**.

```
(define-class Environment Object (next))
```

```
(define-class Full-Environment Environment (variable))
(define (r-extend* r vars)
  (if (pair? vars)
      (r-extend (r-extend* r (cdr vars)) (car vars))
      r ) )
(define (r-extend r var)
  (make-Full-Environment r var) )
(define (find-variable? name r)
  (if (Full-Environment? r)
      (let ((var (Full-Environment-variable r)))
        (if (eq? name
                 (cond ((Variable? var) (Variable-name var))
                       ((Magic-Keyword? var)
                        (Magic-Keyword-name var) ) ) )
            var
            (find-variable? name (Full-Environment-next r)) ) )
      (if (Environment? r)
          (find-variable? name (Environment-next r))
          #f ) ) )
(define (insert-global! variable r)
  (let ((r (find-global-environment r)))
    (set-Environment-next!
     r (make-Full-Environment (Environment-next r) variable) ) ) )
(define (mark-global-preparation-environment g)
  (make-Environment g) )
(define (find-global-environment r)
  (if (Full-Environment? r)
      (find-global-environment (Full-Environment-next r))
      r ) )
```

The way assignment is handled takes the opportunity to annotate all the local variables that will be assigned later; it sets their **mutable?** field to True. That field will be used eventually in Chapter 10. [see p. 359]

```
(define (objectify-assignment variable e r)
  (let ((ov (objectify variable r))
        (of (objectify e r)) )
    (cond ((Local-Reference? ov)
           (set-Local-Variable-mutable?!
            (Local-Reference-variable ov) #t )
           (make-Local-Assignment ov of) )
          ((Global-Reference? ov)
           (make-Global-Assignment (Global-Reference-variable ov) of) )
          (else (objectify-error
                 "Illegal mutated reference" variable )) ) ) )
```

When the initial expression is transformed into an object, it is simple to write an evaluator to interpret this object in a way that verifies the translation procedure. The evaluator resembles those presented in the preceding chapters, especially the one for objects in Chapter 3 crossed with the one for rapid interpretation in Chapter 6 [see p. 183].

Let's take a look at the general outline of this evaluator so that it will be more or less transparent in what follows. Evaluation is managed by the generic function **evaluate**. It takes two arguments: the first is an instance of **Program** and the second an instance of **Environment** (a kind of A-list of pairs made up of variables and values). The initial environment contains only predefined variables. Its static part is the value of **g.predef** while its dynamic part is the value of **sg.predef**. That dynamic part associates instances of **RunTime-Primitive** with predefined functional variables. The execution environment can be extended by **sr-extend**.

9.11.2 Special Forms

The preceding translator doesn't recognize special forms, so for each one of them, we have to indicate how to transform it into an instance of **Program**. We'll associate an appropriate code walker with each keyword. That inspector will simply call one of the preceding functions, all built on the same model.

```
(define special-if
  (make-Magic-Keyword
   'if (lambda (e r)
         (objectify-alternative (cadr e) (caddr e) (cadddr e) r) ) ) )
(define special-begin
  (make-Magic-Keyword
   'begin (lambda (e r)
            (objectify-sequence (cdr e) r) ) ) )
(define special-quote
  (make-Magic-Keyword
   'quote (lambda (e r)
            (objectify-quotation (cadr e) r) ) ) )
(define special-set!
  (make-Magic-Keyword
   'set! (lambda (e r)
           (objectify-assignment (cadr e) (caddr e) r) ) ) )
(define special-lambda
  (make-Magic-Keyword
   'lambda (lambda (e r)
             (objectify-function (cadr e) (cddr e) r) ) ) )
```

Of course, we could define other special forms that we would add to the preceding magic keywords to form the list ***special-form-keywords***.

```
(define *special-form-keywords*
  (list special-quote
        special-if
        special-begin
        special-set!
        special-lambda
        ;; cond, letrec, etc.
        special-let
        ) )
```

9.11.3 Evaluation Levels

We've already seen that endogenous macro expansion hides an evaluator inside. Here we need to remember that the result of objectification is not necessarily evaluated afterwards in the same memory space. In fact, Chapter 10 [see p. 359], about compiling into C, will show just that. Since we don't want to confuse these various evaluators, we'll introduce a tower of evaluators and assume the purest hypothesis, where all the evaluators are distinct: the language, the macro language, the macro language of the macro language, and so forth, will all have different global environments. These evaluators will be represented by instances of the class `Evaluator` which defines their principal characteristics.

```
(define-class Evaluator Object
  ( mother
    Preparation-Environment
    RunTime-Environment
    eval
    expand
    ) )
```

What follows is quite complex. Figure 9.9 illustrates the distinct environments that come into play. The expansion function at one level uses the evaluation function of the next level, but that one itself begins by analyzing the expressions it receives and thus by expanding them. We'll avoid infinite regression that might result from one level expanding an expansion already stripped of macros; in that case, there would be no need of the next level. Generally, two or three levels suffice. Each level has a preparation environment (for **expand**) and an execution environment (for **eval**)—except perhaps the "ground floor" where we use only the expander.

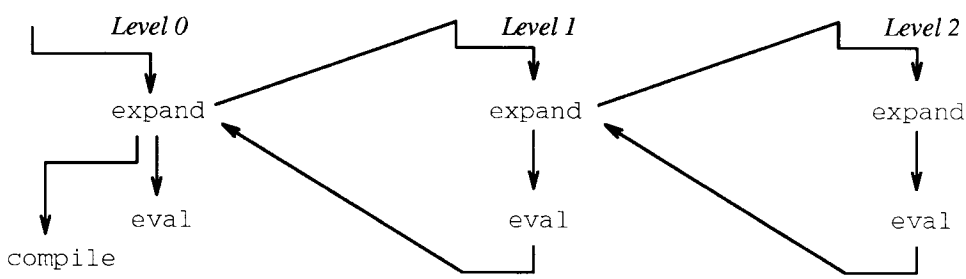


Figure 9.9 Tower of evaluators

The function `create-evaluator` builds the levels of the tower one by one on demand. It builds a new level above the one it receives as an argument. It creates a pair of functions, **expand** and **eval**, along with two environments: preparation and execution. The function **eval** systematically expands its argument before evaluating it; **eval** does that by means of an evaluation engine named **evaluate**. The only thing we have to say about **evaluate** is that it needs only the execution environment, the one saved in the field `RunTime-Environment` of the current instance of `Evaluator`, one that it modifies physically if need be. The expansion func-

tion uses the objectification engine of **objectify** with a preparation environment; that preparation environment is stored in the field **Preparation-Environment** of the current instance of **Evaluator**. To simplify our lives, the global variables discovered during expansion are *ipso facto* created in the associated execution environment by means of the function **enrich-with-new-global-variables!**. The function **eval** is installed reflectively in its own execution environment like a predefined primitive, so there is a different **eval** at each level. The preparation environment is extended by all the special forms accumulated in the variable ***special-form-keywords***. That variable must be visible at every level. The preparation environment is also extended by predefined macros that result from the call to **(make-macro-environment level)**, which we'll document later.

```
(define (create-evaluator old-level)
  (let ((level 'wait)
        (g     g.predef)
        (sg    sg.predef) )
    (define (expand e)
      (let ((prg (objectify
                  e (Evaluator-Preparation-Environment level) )))
        (enrich-with-new-global-variables! level)
        prg ) )
    (define (eval e)
      (let ((prg (expand e)))
        (evaluate prg (Evaluator-RunTime-Environment level)) ) )
    ;; Create resulting evaluator instance
    (set! level (make-Evaluator old-level 'wait 'wait eval expand))
    ;; Enrich environment with eval
    (set! g (r-extend* g *special-form-keywords*))
    (set! g (r-extend* g (make-macro-environment level)))
    (let ((eval-var (make-Predefined-Variable
                     'eval (make-Functional-Description = 1 "") ))
          (eval-fn (make-RunTime-Primitive eval = 1)) )
      (set! g (r-extend g eval-var))
      (set! sg (sr-extend sg eval-var eval-fn)) )
    ;; Mark the beginning of the global environment
    (set-Evaluator-Preparation-Environment!
     level (mark-global-preparation-environment g) )
    (set-Evaluator-RunTime-Environment!
     level (mark-global-runtime-environment sg) )
    level ) )
```

If we need an expander or an evaluator, all we have to do is invoke the function **create-evaluator** and retrieve the **expand** or **eval** function we want simply by reading the fields. Careful: in contrast to a conventional expander, the function **expand** returns an instance of **Program**, not a Scheme expression. A simple conversion function will get us such an expression, if we want it. **[see Ex. 9.4]**

9.11.4 The Macros

According to our plan, predefined macros have already been added to the preparation environment. They were synthesized by the function **make-macro-environ-**

ment. The four predefined macros are **eval-in-abbreviation-world**, **define-abbreviation**, **let-abbreviation**, and **with-aliases**. They all assume the existence of an evaluator belonging to the level above that the function **make-macro-environment** has to create. However, to avoid infinite regression, creating a level above is just a promise that won't be carried out unless the associated evaluator is actually invoked. [**see Ex. 9.3**] Once a supplementary level has been created, it will not be recreated; instead, it endures and accumulates all the definitions that it receives.

```
(define (make-macro-environment current-level)
  (let ((metalevel (delay (create-evaluator current-level))))
    (list (make-Magic-Keyword 'eval-in-abbreviation-world
           (special-eval-in-abbreviation-world metalevel) )
          (make-Magic-Keyword 'define-abbreviation
           (special-define-abbreviation metalevel))
          (make-Magic-Keyword 'let-abbreviation
           (special-let-abbreviation metalevel))
          (make-Magic-Keyword 'with-aliases
           (special-with-aliases metalevel) ) ) ) )
```

The macro **eval-in-abbreviation-world** is the simplest because it merely evaluates its body with the evaluator from the level above. It demands that the level above be constructed only if need be. Like any macro, the result submitted anew to the objectification function of the current level.

```
(define (special-eval-in-abbreviation-world level)
  (lambda (e r)
    (let ((body (cdr e)))
      (objectify ((Evaluator-eval (force level))
                  `(,special-begin . ,body) )
                 r ) ) ) )
```

The macro **define-abbreviation** creates and modifies global macros. The associated expander is created and then evaluated at the level above; a new magic keyword is created in the global preparation environment of the current level. The function **invoke** is a different means from **evaluate** for starting the evaluation engine. When a macro is invoked by **invoke**, it begins its calculations in the execution environment of the level above, wrapped up in the closure of the expander. The result of such a macro is, here, **#t**. We could have returned the name of the macro being defined, but that would have meant a few bytes of extra garbage (that is, a symbol).

```
(define (special-define-abbreviation level)
  (lambda (e r)
    (let* ((call      (cadr e))
           (body      (cddr e))
           (name      (car call))
           (variables (cdr call)) )
      (let ((expander ((Evaluator-eval (force level))
                       `(,special-lambda ,variables . ,body) )))
        (define (handler e r)
          (objectify (invoke expander (cdr e)) r) )
        (insert-global! (make-Magic-Keyword name handler) r)
```

```
        (objectify #t r) ) ) ) )
```

We can create local macros in a similar way. The difference is that these magic keywords are inserted in front of the preparation environment at the current level in order to maintain local scope.

```
(define (special-let-abbreviation level)
  (lambda (e r)
    (let ((level  (force level))
          (macros (cadr e))
          (body   (cddr e)) )
      (define (make-macro def)
        (let* ((call      (car def))
               (body      (cdr def))
               (name      (car call))
               (variables (cdr call)) )
          (let ((expander ((Evaluator-eval level)
                           `(,special-lambda ,variables . ,body) )))
            (define (handler e r)
              (objectify (invoke expander (cdr e)) r) )
            (make-Magic-Keyword name handler) ) ) )
      (objectify `(,special-begin . ,body)
                 (r-extend* r (map make-macro macros)) ) ) ) )
```

The most complex of the four predefined macros is **with-aliases** because it has multiple effects. It must capture the meaning of a number of words in the current expander. It must bind those meanings to variables for the evaluator at the level above during the expansion of its body. This interaction among the scope, the duration, and the levels makes this one so complex.

```
(define (special-with-aliases level)
  (lambda (e current-r)
    (let* ((level   (force level))
           (oldr    (Evaluator-Preparation-Environment level))
           (oldsr   (Evaluator-RunTime-Environment level))
           (aliases (cadr e))
           (body    (cddr e)) )
      (let bind ((aliases aliases)
                 (r       oldr)
                 (sr      oldsr) )
        (if (pair? aliases)
            (let* ((variable (car (car aliases)))
                   (word     (cadr (car aliases)))
                   (var      (make-Local-Variable variable #f #f)) )
              (bind (cdr aliases)
                    (r-extend r var)
                    (sr-extend sr var (objectify word current-r)) ) )
            (let ((result 'wait))
              (set-Evaluator-Preparation-Environment! level r)
              (set-Evaluator-RunTime-Environment! level sr)
              (set! result (objectify `(,special-begin . ,body)
                                      current-r ))
              (set-Evaluator-Preparation-Environment! level oldr)
```

```
(set-Evaluator-RunTime-Environment! level oldsr)
result ) ) ) ) ) )
```

Modifying the execution environment of the level above would be better accommodated by a dynamic binding in order to be sure that the right execution environment will be restored after the expansion of the body of the form `with-aliases`.

The variables bound by `with-aliases` have as their values the results of the function `objectify`, that is, instances of `Program` or `Magic-Keyword`. Since these instances can appear in the result of a macro expansion, the function `objectify` must be able to recognize these cases in the expressions that it handles. For that reason, the function `objectify` begins by testing whether the expression it received is a magic keyword or a program already objectified. [see p. 345]

9.11.5 Limits

Following the rules of macro hygiene lets us play around statically with words and bindings, to capture a meaning and use it in contexts where it would not be visible otherwise, even in places where it should not be usable otherwise. For example, we can write this:

```
(let ((count 0))
  (with-aliases ((c count))
    (define-abbreviation (tick) c) )
  (tick) )
(let ((count 1)(c 2))
  (tick) )
```

The global macro `tick` refers to the local variable `count` which is no longer visible from the second calling site of the macro `tick`. Indeed, we get a new kind of error that way: a reference to a non-existing variable, even if a variable of the same name (like `count`) appears in the calling environment of `tick`! You can see the same thing again more clearly in the equivalent "de-objectified" form:

```
((LAMBDA (COUNT501)
  (BEGIN #T ;;(tick)⇝ count501
         COUNT501 ))
 0 )
((LAMBDA (COUNT502 C503) COUNT501) 1 2)
```

We have to choose the position of `with-aliases` carefully because it is a kind of `let` for the evaluator at the level above. If we wrote this:

```
(define-abbreviation (loop . body)
  (with-aliases ((cc call/cc)(lam lambda) (ll let))
    (let ((loop (gensym)))
      `(,cc (,lam (exit) (,ll ,loop () ,@body (,loop)))) ) ) )
```

then the meanings captured here are caught at the level of the macro definition, and thus they are valid only in the world of macros of macros. In practice, `with-aliases` appears in `define-abbreviation` and is thus evaluated in the world of macros when the abbreviation `loop` gets into the normal world. The variable `cc` is bound in the macro world to the meaning that the word `call/cc` has there. A call to the macro `loop` will lead to an error of the type "`cc: unknown variable`."

With the exception of keywords for special forms or other keywords like **else** or **=>**, the meaning of local or global variables could have been captured by first class environments and the special forms **import** and **export**. [see p. 296] However, here the mechanism we used is more powerful since it can also capture the essence of keywords for special forms and especially because it is static rather than dynamic.

The macro mechanism we've defined here allows predefined macros to be written, ones that the end user could not write, like, for example, macros creating global variables by direct use of the function **insert-global!**.

Nevertheless, this system has its imperfections. If the user wants to run through the macro expanded expressions, he or she will encounter at least two problems. First, the function **expand** has not been made visible. One way of making it visible is for **expand** at one level to be the value of a variable at the next higher level so that **(eval-in-abbreviation-world (expand 'ε))** would be identical to ε. Second, we have to know the structure of subclasses of **Program** if we want to walk through these objects.

One of the goals of this system of macros was to show that hygienic macro expansion and compilation are tightly linked since they take advantage of an important common basis. In the preceding code, if we take out those parts connected with evaluation and with objectification strictly speaking, there is nothing left that depends specifically on macro expansion except the function **objectify**, the function **objectify-symbol**, and the functions connected to the four predefined macros. That's only about a hundred lines—very little, in fact.

A real macro system would have many other details to pin down:

1. how to program essential syntax, like **or**, **and**, **letrec**, internal **define**, etc.;
2. how to program the notation for *backquote*, intertwined as it is with expansion and objectification;
3. how to allow macro-symbols in order to resolve the problem we saw earlier in **define-handy-method**. [see p. 340]

However, we did attain the goal we first set: to introduce the idea of capturing a meaning.

9.12 Conclusions

We can summarize the problems connected to macros in two points: they are indispensible, but there are no two compatible systems. They exist in practice in many possible and divergent implementations. We've tried to survey the entire range. Few reference manuals for Lisp or Scheme indicate precisely which model for macros they provide, but in their defense we have to admit that this chapter is probably the first document to attempt to describe the immense variety of possible macro behavior.

9.13 Exercises

Exercise 9.1 : Define the macro **repeat** given at the beginning of this chapter. Make it hygienic.

Exercise 9.2 : Use `define-syntax` to define a macro taking a sequence of expressions as its argument and printing their numeric order between them. For example, (`enumerate` π_1 π_2 ... π_n) should expand into something that when evaluated will print 0, then calculate π_1, print 1, calculate π_2, etc.

Exercise 9.3 : Modify the macro system to implement the variation corresponding to a uniquely unique world.

Exercise 9.4 : Write a function to convert an instance of `Program` into an equivalent S-expression.

Exercise 9.5 : Study the programs that define MEROONET [see p. 417] to see what belongs to the expansion library, what to the execution library, what to both.

Recommended Reading

In [Gra93], you will find a very interesting apology for macros. There are more theoretical articles like [QP90]. Others, like [KFFD86, DFH88, CR91a, QP91b], focus more on the problems of hygienic expansion. There is a new and promising model of expansion in [dM95].

10
Compiling into C

ONCE again, here's a chapter about compilation, but this time, we'll look at new techniques, notably, flat environments, and we have a new target language: C. This chapter takes up a few of the problems of this odd couple. This strange marriage has certain advantages, like free optimizations of the compilation at a very low level or freely and widely available libraries of immense size. However, there are some thorns among the roses, such as the fact that we can no longer guarantee tail recursion, and we have a hard time with garbage collection.

Compiling into a high-level language like C is interesting in more ways than one. Since the target language is so rich, we can hope for a translation that is closer to the original than would be some shapeless, linear salmagundi. Since C is available on practically any machine, the code we produce has a good chance of being portable. Moreover, any optimizations that such a compiler can achieve are automatically and implicitly available to us. This fact is particularly important in the case of C, where there are compilers that carry out a great many optimizations with respect to allocating registers, laying out code, or choosing modes of address—all things that we could ignore when we focused on only one source language.

On the other hand, choosing a high-level language as the target imposes certain philosophic and pragmatic constraints as well. Such a language is typically designed for a particular style of program without supposing that such programs might be generated by other programs. In consequence, certain limits, like at most 32 arguments in function calls, or less than 16 levels of lexical blocks, and so forth, may be almost tolerable for normal users, but they are quite problematic for automatically generated programs. It is not unusual for a problem blessed with a few macros to multiple its size by 5, 10, or 20 times when it is translated into C. Such an increase can pose problems for a compiler unaccustomed to such monsters.

Moreover, the execution model of the target language may have little to do with the execution model of the source language, and that, too, can limit or complicate the translation from one to the other. C was designed as a language for writing an operating system (namely, UN⋆X), so by deliberate policy, it explicitly manages memory. That fact can lead to all sorts of excesses, such as pointers running amuck in the sense that the programmer loses all control over them—a situation that does not arise with Lisp, which is a safe language in that respect.

In addition, C is not particularly adapted to writing functional programs nor recursive programs either because calling a function there is notoriously expensive. Programmers generally dislike its slowness for that and consequently use it as little as possible, a practice that justifies the implementers in not trying to improve it since they know that programmers seldom use it because it's slow, etc.

Be that as it may, compiling into C is in fashion now, as witnessed by Kyoto COMMON LISP in [YH85], and refined somewhat in AKCL by William Schelter, or WCL in [Hen92b], or CLICC in [Hof93], or EcoLisp in [Att95], or Scheme→C in [Bar89], or Sqil in [Sén91], or ILOG TALK in [ILO94], or Bigloo in [Ser94].

What we'll present is no rival to those. It's just a skeleton of a compiler, but it will suffice to show you a great number of interesting points. A simple solution (some would even say a trivial one) would be to change the byte-code generator we saw in Chapter 7 [see p. 223] just enough to make it generate C. Each instruction byte could be expanded into a few appropriate C instructions. However, we're going to take a completely different path, using the technique of flat environments that we alluded to earlier. [see p. 202] This new compiler will be partitioned into passes, most of which will be produced by specialized code walkers. This technique is based on a systematic use of objects to represent and transform the code to compile. In fact, we think this systematic use of objects will make this chapter particularly elegant and easy to extend.

10.1 Objectification

We've already presented a converter from programs into objects in Section 9.11.1, [see p. 344]. It takes care of any possible macro expansions that might be found and it returns an instance of the class **Program**. To fill out that curt description, we offer the illustration in Figure 10.1 of one result of that phase of objectification. In that illustration, we'll use the following expression, which contains at least one example of every aspect we studied. For the sake of brevity, the names of classes have been abbreviated; only one part of the example appears in the figure; lists have been put into giant parentheses. We'll come back to this example in what follows.

```
(begin
  (set! index 1)
  ((lambda (cnter . tmp)
     (set! tmp (cnter (lambda (i) (lambda x (cons i x)))))
     (if cnter (cnter tmp) index) )
   (lambda (f)
     (set! index (+ 1 index))
     (f index) )
   'foo ) )   → (2 3)
```

10.2 Code Walking

Code walking, as in [Cur89, Wat93], is an important technique. It consists of traversing a tree that represents a program to compile and enriching it with various

annotations to prepare the ultimate phase, that is, code generation. Depending on what we want to establish, we may favor various schemes for traversing the tree and various ways of collecting information. In short, there is no universal code walker! The evaluators in this book are rather special code walkers; so are precompilation functions like **meaning** in the chapter about fast interpretation [see p. 207].

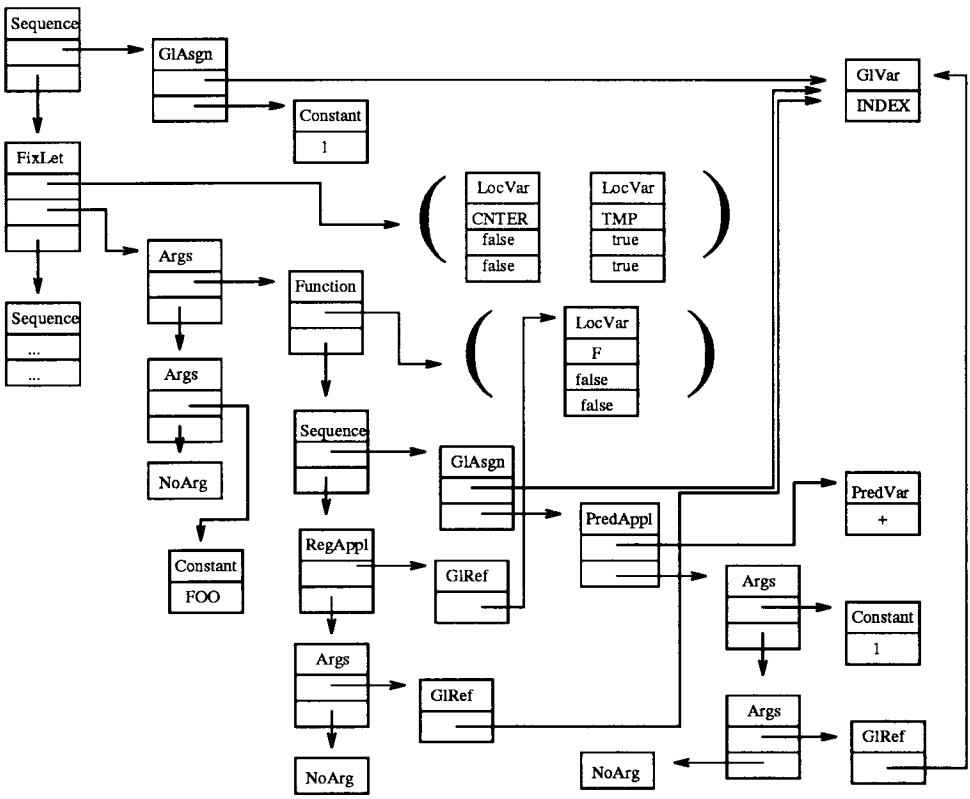


Figure 10.1 Objectified code

We're going to define only one code walker; it systematically modifies the tree that we provide it. To begin, it will be an excellent example of a *metamethod*. The function **update-walk!** takes these arguments: a generic function, an object of the class **Program**, and possibly supplementary arguments. It replaces each field of the object containing an instance of the class **Program** by the result of applying that generic function to the value of that field. Its return value is the initial object. To do all that, each object is inspected; the fields of its class are extracted one by one, verified, and recursively analyzed to see whether they belong to the class **Program**. You can see now why the class **Arguments** (which of course represents the arguments of an instance of **Application**) inherit from the class **Program**: they have to be available for inspection by a code walker.

```
(define (update-walk! g o . args)
  (for-each (lambda (field)
```

```
            (let ((vf (field-value o field)))
              (when (Program? vf)
                (let ((v (if (null? args) (g vf)
                             (apply g vf args) )))
                  (set-field-value! o v field) ) ) ) )
          (Class-fields (object->class o)) )
  o )
```

Programming like that may seem simplistic to you, but it's highly convenient, as we'll prove in the following sections, where our program will metamorphose very efficiently. It's obvious that some of the passes for various transformations could be combined and thus speeded up. We'll avoid that temptation so that we can clearly separate the effects of these transformations.

10.3 Introducing Boxes

We'll suppress all local assignments in favor of functions operating on boxes. You've already seen this transformation earlier in this book [see p. 114]. This transformation will also be useful as our first example of code walking.

What we want is to replace all the assignments of local variables by writing in boxes. We also have to be sure that reading variables is transformed into reading in boxes. If we take into account the interface to the code walker, we must provide it with a generic function carrying out this work. By default, this generic function recursively invokes the code walker for the discriminating object. The interaction between the generic function and the code walker is the chief strength of this union.

```
(define-generic (insert-box! (o Program))
  (update-walk! insert-box! o) )
```

We'll introduce three new types of syntactic nodes to represent programs transformed in this way. We'll document them as part of the transformations that need them.

```
(define-class Box-Read Program (reference))
(define-class Box-Write Program (reference form))
(define-class Box-Creation Program (variable))
```

Quite fortunately for us, during objectification, we've taken care to mark all the mutable local variables by their field **mutable?**. Consequently, transforming a read of a mutable variable into an instance of **Box-Read** is easy.

```
(define-method (insert-box! (o Local-Reference))
  (if (Local-Variable-mutable? (Local-Reference-variable o))
      (make-Box-Read o)
      o ) )
```

Transforming assignments is equally easy. A local assignment is transformed into an instance of **Box-Write**. However, we must take into account the structure of the algorithm for code walking, so we must not forget to call the code walker recursively on the form providing the new value of the assigned variable.

```
(define-method (insert-box! (o Local-Assignment))
  (make-Box-Write (Local-Assignment-reference o)
```

```
                    (insert-box! (Local-Assignment-form o)) ) )
```

Once every access (whether read or write) to mutable variables has been transformed to occur within boxes, the only thing left to do is to create those boxes. Local mutable variables can be created only by `lambda` or `let` forms, that is, by `Function` or `Fix-Let`[1] nodes. The technique is to insert an appropriate way to "put it in a box" in front of the body of such forms. [see p. 114] So here is how to specialize the function `insert-box!` for the two types of nodes that might introduce mutable local variables. They both depend on a subfunction to insert as many instances of `Box-Creation` in front of their body as there are mutable local variables for which boxes must be allocated.

```
(define-method (insert-box! (o Function))
  (set-Function-body!
   o (insert-box!
      (boxify-mutable-variables (Function-body o)
                                (Function-variables o) ) ) )
  o )
(define-method (insert-box! (o Fix-Let))
  (set-Fix-Let-arguments! o (insert-box! (Fix-Let-arguments o)))
  (set-Fix-Let-body!
   o (insert-box!
      (boxify-mutable-variables (Fix-Let-body o)
                                (Fix-Let-variables o) ) ) )
  o )
(define (boxify-mutable-variables form variables)
  (if (pair? variables)
      (if (Local-Variable-mutable? (car variables))
          (make-Sequence
           (make-Box-Creation (car variables))
           (boxify-mutable-variables form (cdr variables)) )
          (boxify-mutable-variables form (cdr variables)) )
      form ) )
```

With that, the transformation is complete and has been specified in only four methods, focused solely on the forms that are important for transformations. You can see the result in Figure 10.2, representing only the subparts of the previous figure that have changed.

10.4 Eliminating Nested Functions

As a language, C does not support functions inside other functions. In other words, a `lambda` inside another `lambda` cannot be translated directly. Consequently, we must eliminate these cases in a way that turns the program to compile into a simple set of closed functions, that is, functions without free variables. Once again, we're lucky to find such a natural transformation. We'll call it `lambda`-*lifting* because it makes `lambda` forms migrate toward the exterior in such a way that there are no remaining `lambda` forms in the interior. Many variations on this transformation

1. Even though we have not identified reducible applications, we could at least have a method to handle them.

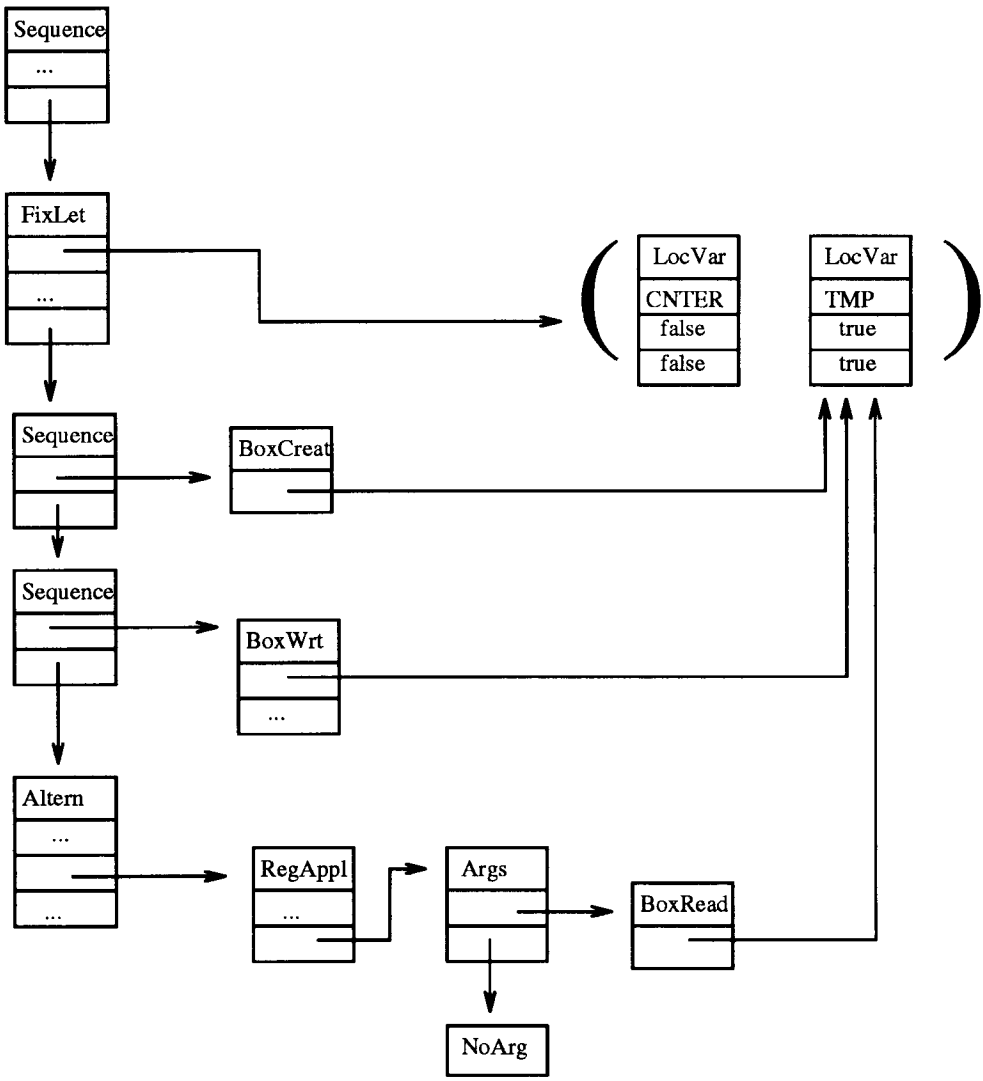


Figure 10.2 `(lambda (cnter . tmp) (set! tmp ... ) (if ... (...tmp) ...))`

are possible, depending on the qualities we want to preserve or promote, as in [WS94, KH89, CH94].

Evaluating a **lambda** form leads to synthesizing a closure, a kind of record that encloses its definition environment. When a closure is invoked, a special function (that we have named **invoke**) knows how to evaluate the body of this closure by providing it the means to retrieve the values of free variables present in the body. In fact, only the invoker knows how to invoke closures, so we can replace these closures by records without disturbing the rest of the program on condition that every functional application passes by the generic invoker, **invoke**, the one that knows how to do this.

Let's look at an example illustrating the variation known as *OO-lifting* in [Que94]. As usual, we'll use our familiar guinea pig, the factorial function, like this:

```
(define (fact n k)
  (if (= n 0) (k 1)
      (fact (- n 1) (lambda (r) (k (* n r)))) ) )
```

We can translate it to eliminate the internal **lambda** form enclosing the variables **n** and **k**, like this:

```
(define-class Fact-Closure Object (n k))
(define-method (invoke (f Fact-Closure) r)
  (invoke (Fact-Closure-k f) (* (Fact-Closure-n f) r)) )
(define (fact n k)
  (if (= n 0) (invoke k 1)
      (fact (- n 1) (make-Fact-Closure n k)) ) )
```

The essence of the transformation is to replace the way the closure is built by an allocation of an object (**make-Fact-Closure**) containing the free variables of the body of the closure (here, **n** and **k**). A particular class is associated with this object (here, **Fact-Closure**) so that **invoke** can determine which method to use when this object is invoked.

The basis of the transformation is, for each abstraction (**Function**), to calculate the set of free variables present in its body. A new class—**Flat-Function**, refining **Function**—will store that information. The collection of free variables will be represented by instances of the class **Free-Environment** and will be ended by **No-Free**. These two classes are similar to the classes **Arguments** and **No-Argument**: they also derive from **Program**, since they also represent terms that can be evaluated. We'll also identify references to these free variables by a particular class of reference: **Free-Reference**.

```
(define-class Flat-Function Function (free))
(define-class Free-Environment Program (first others))
(define-class No-Free Program ())
(define-class Free-Reference Reference ())
```

To make this transformation work, the code walker will help out again. The function **lift!** is the interface for the entire treatment. The generic function **lift-procedures!** serves as the argument to the code walker. It takes two supplementary arguments itself: the abstraction **flatfun** in which the free variables

are stored and the list of current bound variables in the variable **vars**. By default, the code walker is called recursively for all the instances of **Program**.

```
(define (lift! o)
  (lift-procedures! o #f '()) )
(define-generic (lift-procedures! (o Program) flatfun vars)
  (update-walk! lift-procedures! o flatfun vars) )
```

Only three methods are needed for the transformation. The first identifies the references to free variables, transforms them, and collects them in the current abstraction. The function **adjoin-free-variable** adds any free variable that is not yet one of the free variables already identified in the current abstraction.

```
(define-method (lift-procedures! (o Local-Reference) flatfun vars)
  (let ((v (Local-Reference-variable o)))
    (if (memq v vars)
        o (begin (adjoin-free-variable! flatfun o)
                 (make-Free-Reference v) ) ) ) )
(define (adjoin-free-variable! flatfun ref)
  (when (Flat-Function? flatfun)
    (let check ((free* (Flat-Function-free flatfun)))
      (if (No-Free? free*)
          (set-Flat-Function-free!
           flatfun (make-Free-Environment
                    ref (Flat-Function-free flatfun) ) )
          (unless (eq? (Reference-variable ref)
                       (Reference-variable
                        (Free-Environment-first free*) ) )
            (check (Free-Environment-others free*)) ) ) ) ) )
```

The form **Fix-Let** creates new bindings that might hide others. Consequently, we must add the variables of **Fix-Let** to the current bound variables before we analyze its body. At that point, it is very useful to save the reducible applications as such because it would be very costly to build these closures at execution time.

```
(define-method (lift-procedures! (o Fix-Let) flatfun vars)
  (set-Fix-Let-arguments!
   o (lift-procedures! (Fix-Let-arguments o) flatfun vars) )
  (let ((newvars (append (Fix-Let-variables o) vars)))
    (set-Fix-Let-body!
     o (lift-procedures! (Fix-Let-body o) flatfun newvars) )
    o ) )
```

Finally, the most complicated case is how to treat an abstraction. The body of the abstraction will be analyzed, and an instance of **Flat-Function** will be allocated to serve as the receptacle for any free variables discovered there. Since the free variables found there can also be free variables in the surrounding abstraction, the code walker is started again on this list of free variables which have been cleverly coded as a subclass of **Program**.

```
(define-method (lift-procedures! (o Function) flatfun vars)
  (let* ((localvars (Function-variables o))
         (body    (Function-body o))
         (newfun (make-Flat-Function localvars body (make-No-Free))) )
```

```
(set-Flat-Function-body!
 newfun (lift-procedures! body newfun localvars) )
(let ((free* (Flat-Function-free newfun)))
  (set-Flat-Function-free!
   newfun (lift-procedures! free* flatfun vars) ) )
newfun ) )
```

As usual, we'll show you the partial effect of this transformation on the current example in Figure 10.3.

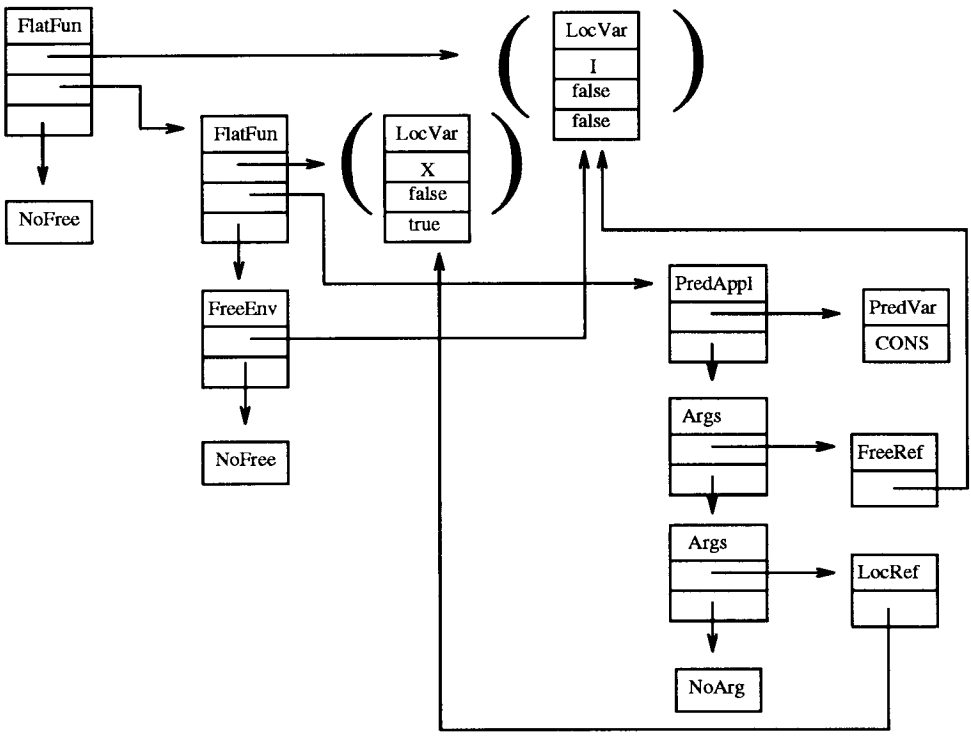


Figure 10.3 `(lambda (i) (lambda x ... ))`

10.5 Collecting Quotations and Functions

The preceding transformation left functions in place since syntactically there were still closed `lambda` forms (that is, ones with no more free variables) inside other `lambda` forms. In doing that, however, we were merely temporizing to get a jump on the problem! The next code walker will extract quotations and definitions of functions from a program in order to put them at a higher level. We'll need only two special methods to do that.

The function `extract-things!` will transform a program into an instance of `Flattened-Program`, a specialization of the class `Program`, one provided with three supplementary fields: `form` containing the program to evaluate; `quotations`,

the list of quotations, of course; and **definitions**, the list of function definitions. Quotations will be handled by references to global variables of a new class: **Quotation-Variable**. Functions will be organized by their order number, an integer, **index**. Finally, the creation of a closure will be translated by a new syntactic node: **Closure-Creation**. (Very soon a lot of details will all become clearer at the same time.)

```
(define-class Flattened-Program Program (form quotations definitions))
(define-class Quotation-Variable Variable (value))
(define-class Function-Definition Flat-Function (index))
(define-class Closure-Creation Program (index variables free))
```

Strictly speaking, we have to say that the code walker is the result of interaction between **extract-things!** and the generic function **extract!**. In order to eliminate every recourse to a global variable, (for example, so we can compile programs in parallel), the results of the code walker will be inserted in the final object that is passed as a supplementary argument to the generic function.

```
(define (extract-things! o)
  (let ((result (make-Flattened-Program o '() '())))
    (set-Flattened-Program-form! result (extract! o result))
    result ) )
(define-generic (extract! (o Program) result)
  (update-walk! extract! o result) )
```

Quotations are simply accumulated in the field for quotations of the final program; they are replaced by references to global variables initialized with these quotations.

```
(define-method (extract! (o Constant) result)
  (let* ((qv* (Flattened-Program-quotations result))
         (qv  (make-Quotation-Variable (length qv*)
                                       (Constant-value o) )) )
    (set-Flattened-Program-quotations! result (cons qv qv*))
    (make-Global-Reference qv) ) )
```

We search for abstractions in the nodes of the class **Flat-Function**; at the same time, these nodes are transformed into instances of **Closure-Creation**. You might also imagine sharing those same abstractions; that is, we could make **adjoin-definition!** a memo-function. One point that might seem strange is that when we build the closure, we save the list of variables from the original abstraction. We will save it to make the computation of the arity of the closure easier when we generate C.

```
(define-method (extract! (o Flat-Function) result)
  (let* ((newbody   (extract! (Flat-Function-body o) result))
         (variables (Flat-Function-variables o))
         (freevars  (let extract ((free (Flat-Function-free o)))
                      (if (Free-Environment? free)
                          (cons (Reference-variable
                                 (Free-Environment-first free) )
                                (extract
                                 (Free-Environment-others free) ) )
                          '() ) ))
```

```
        (index (adjoin-definition!
                result variables newbody freevars )) )
    (make-Closure-Creation index variables (Flat-Function-free o)) ) )
(define (adjoin-definition! result variables body free)
  (let* ((definitions (Flattened-Program-definitions result))
         (newindex (length definitions)) )
    (set-Flattened-Program-definitions!
     result (cons (make-Function-Definition
                   variables body free newindex )
                  definitions ) )
    newindex ) )
```

Again, to illustrate the results of this transformation, you can see a few choice extracts from the example in Figure 10.4.

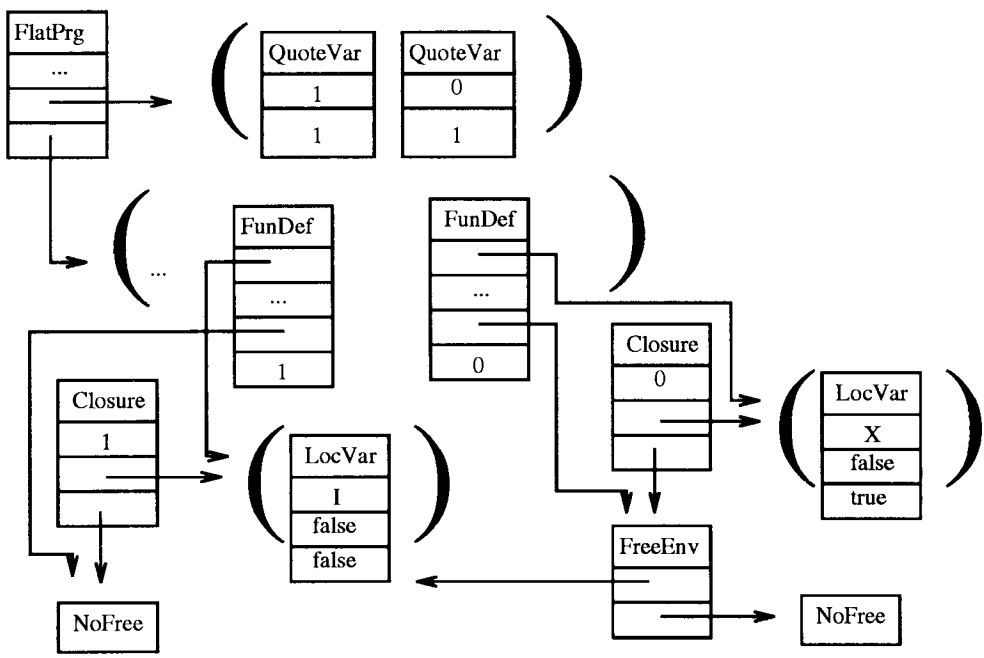


Figure 10.4 `(begin (set! ... ) ((lambda ... ) ... ))`

Finally, we are going to convert the entire program into the application of a closure. That is, π will be transformed into `((lambda () `π`))`.

```
(define (closurize-main! o)
  (let ((index (length (Flattened-Program-definitions o))))
    (set-Flattened-Program-definitions!
     o (cons (make-Function-Definition
              '() (Flattened-Program-form o) '() index )
             (Flattened-Program-definitions o) ) )
    (set-Flattened-Program-form!
     o (make-Regular-Application
        (make-Closure-Creation index '() (make-No-Free))
```

```
      (make-No-Argument) ) )
  o ) )
```

10.6 Collecting Temporary Variables

We have a decisive advantage over you, the reader, because we know already what we want to generate. Hoping to be original, we have chosen to convert Scheme expressions into C expressions. Deciding to do that might seem eccentric since C is a language that involves instructions, but our choice lets us respect the structure of Scheme. One problem then is that nodes of type `Fix-Let` cannot be translated into C because C does not have expressions that let us introduce local[2] variables. Our solution is to collect all the local variables from the `Fix-Let` forms that are introduced, C function by C function. (This point will be clearer soon.) [see p. 370]

For that reason, we'll introduce a new code walker. Its goal is to take a census of all the local variables from internal nodes of type `Fix-Let` and then rename them. This α-conversion will eliminate any potential name conflicts. The temporary variables are accumulated in a new specialization of `Function-Definition`, namely, the class `With-Temp-Function-Definition`.

```
(define-class With-Temp-Function-Definition Function-Definition
  (temporaries) )
```

The function `gather-temporaries!` implements that transformation. It will use the generic function `collect-temporaries!` in concert with the code walker. The second argument of that generic function will be the place where temporary local variables are stored. The third argument stores the list associating old names of variables with their new names so that the renaming can be carried out.

```
(define (gather-temporaries! o)
  (set-Flattened-Program-definitions!
   o (map (lambda (def)
            (let ((flatfun (make-With-Temp-Function-Definition
                            (Function-Definition-variables def)
                            (Function-Definition-body def)
                            (Function-Definition-free def)
                            (Function-Definition-index def)
                            '() )))
              (collect-temporaries! flatfun flatfun '()) ) )
          (Flattened-Program-definitions o) ) )
  o )
(define-generic (collect-temporaries! (o Program) flatfun r)
  (update-walk! collect-temporaries! o flatfun r) )
```

To achieve all our wishes, we need only three more methods. Local references are renamed, if need be, and we must not forget to rename any variables that we put into boxes.

```
(define-method (collect-temporaries! (o Local-Reference) flatfun r)
  (let* ((variable (Local-Reference-variable o))
```

2. However, gcc offers such an expression in the syntactic form ({ ... }).

```
        (v (assq variable r)) )
    (if (pair? v) (make-Local-Reference (cdr v)) o) ) )
(define-method (collect-temporaries! (o Box-Creation) flatfun r)
  (let* ((variable (Box-Creation-variable o))
         (v (assq variable r)) )
    (if (pair? v) (make-Box-Creation (cdr v)) o) ) )
```

The most complicated method, of course, is the one involved with `Fix-let`. It is recursively invoked on its arguments, and then it renames its local variables by means of the function `new-renamed-variable`. It also adds those new variables to the current function definition and is finally recursively invoked on its body in the appropriate new substitution environment.

```
(define-method (collect-temporaries! (o Fix-Let) flatfun r)
  (set-Fix-Let-arguments!
   o (collect-temporaries! (Fix-Let-arguments o) flatfun r) )
  (let* ((newvars (map new-renamed-variable
                       (Fix-Let-variables o) ))
         (newr (append (map cons (Fix-Let-variables o) newvars) r)) )
    (adjoin-temporary-variables! flatfun newvars)
    (set-Fix-Let-variables! o newvars)
    (set-Fix-Let-body!
     o (collect-temporaries! (Fix-Let-body o) flatfun newr) )
    o ) )
(define (adjoin-temporary-variables! flatfun newvars)
  (let adjoin ((temps (With-Temp-Function-Definition-temporaries
                       flatfun ))
               (vars newvars) )
    (if (pair? vars)
        (if (memq (car vars) temps)
            (adjoin temps (cdr vars))
            (adjoin (cons (car vars) temps) (cdr vars)) )
        (set-With-Temp-Function-Definition-temporaries!
         flatfun temps ) ) ) )
```

When variables are renamed, a new class of variables comes into play along with a counter to number them sequentially.

```
(define-class Renamed-Local-Variable Variable (index))
(define renaming-variables-counter 0)
(define-generic (new-renamed-variable (variable)))
(define-method (new-renamed-variable (variable Local-Variable))
  (set! renaming-variables-counter (+ renaming-variables-counter 1))
  (make-Renamed-Local-Variable
   (Variable-name variable) renaming-variables-counter ) )
```

10.7 Taking a Pause

The following Scheme expression textually suggests the final result of the metamorphoses that our example has submitted to so far. That is,

1. We introduced boxes.

2. We eliminated nested functions.
3. We collected quotations and function definitions.
4. We gathered up the temporary variables.

```
(define quote_5 'foo)                    ;collected quotation
(define-class Closure_0 Object ())       ;abstraction (lambda (f) ...)
(define-method (invoke (self Closure_0) f)
  (begin
    (set! index (+ 1 index))
    (invoke f index) ) )                 ;calculated call
(define-class Closure_1 Object (i))      ;abstraction (lambda x ...)
(define-method (invoke (self Closure_1) . x)
  (cons (Closure_1-i self)               ;closed variable i
        x ) )
(define-class Closure_2 Object ())       ;abstraction (lambda (i) ...)
(define-method (invoke (self Closure_2) i)
  (make-Closure_1 i) )                   ;allocation of a closure
(define-class Closure_3 Object ())       ;abstraction (lambda ()program)
(define-method (invoke (self Closure_3))
  ((lambda (cnter_1 tmp_2)               ;renaming local variables
     (set! index 1)
     (set! cnter_1 (make-Closure_0))     ;initializing a temporary variable
     (set! tmp_2 (cons quote_5 '()))
     (set! tmp_2 (make-box tmp_2))       ;putting it into a box
     (box-write! tmp_2                   ;mutable variable
                 (invoke cnter_1 (make-Closure_2)) )
     (if cnter_1 (invoke cnter_1 (box-read tmp_2)) index) ) ) )
(invoke (make-Closure_3))                ;invocation of main program
```

10.8 Generating C

Now we've actually arrived on the enchanted shores of code generation, and as it happens, generating C. We're actually ready. There are still just a few more explanations needed. Your author does not pretend to be an expert in C, and in fact, he owes what he knows about C to careful reading in such sources as [ISO90, HS91, CEK+89]. He assumes that you've at least heard of C, but you're not encumbered by any preconceived notion of what it is exactly.

The abstract syntactic tree is complete and just waiting to be compiled, like by a *pretty-printer*, into the C language. Code generation is simple since it is so high-level. The function **compile->C** takes an S-expression, applies the set of transformations we just described to it in the right order, and eventually generates the equivalent program on the output port, **out**.

```
(define (compile->C e out)
  (set! g.current '())
  (let ((prg (extract-things! (lift! (Sexp->object e)))))
    (gather-temporaries! (closurize-main! prg))
    (generate-C-program out e prg) ) )
```

```
(define (generate-C-program out e prg)
  (generate-header             out e)
  (generate-global-environment out g.current)
  (generate-quotations         out (Flattened-Program-quotations prg))
  (generate-functions          out (Flattened-Program-definitions prg))
  (generate-main               out (Flattened-Program-form prg))
  (generate-trailer            out)
  prg )
```

Like any other C program, and more generally like some animated creature, this one has a beginning, middle, and end. So that we can have a trace, the compiled expression is printed nicely as a comment (by the function `pp` which is not standard in Scheme) A directive to the C preprocessor includes a standard header file, `scheme.h`, to define all we need for the rest of the compilation. From now on, we will also use the usual non-standard function `format` to print C code.

```
(define (generate-header out e)
  (format out "/* Compiler to C $Revision: 4.1 $ ~%")
  (pp e out)                                ; DEBUG
  (format out "  */~%~%#include \"scheme.h\"~%") )
(define (generate-trailer out)
  (format out "~%/* End of generated code. */~%") )
```

Now things get complicated. The compiled result of our current example appears on page 388. You might want to look at it before you read further.

10.8.1 Global Environment

The global variables of the program to compile fall into two categories: predefined variables, like `car` or `+`, and mutable global variables. We assume that predefined global variables cannot be changed and belong to a library of functions that will be linked to the program to make an executable. In contrast, we have to generate the global environment of variables that can be modified. In other words, we will exploit the contents of `g.current` where we accumulated the mutable global variables that appeared as free variables in the program submitted to the compiler.

The generation that we're tending towards depends on the fact that we compile an entire program, not just a fragment expecting separate compilation. (Separate compilation raises problems that we simply did not have space to cover here.) **[see p. 260]**

In order to simplify the C code that we generate, we will use C macros to make the code more readable. For example, we will declare a global variable in C by means of the macro `SCM_DefineGlobalVariable`. The second argument of that macro is the character string corresponding to the original name[3] in Scheme. It can be used during debugging.

```
(define (generate-global-environment out gv*)
  (when (pair? gv*)
```

3. The `read` function used to read the examples to compile in this chapter converts all the names of symbols in upper case.

```
    (format out "~%/* Global environment: */~%")
    (for-each (lambda (gv) (generate-global-variable out gv))
              gv* )) )
(define (generate-global-variable out gv)
  (let ((name (Global-Variable-name gv)))
    (format out "SCM_DefineGlobalVariable(~A,\"~A\");~%"
            (IdScheme->IdC name) name ) ) )
```

There you can see the first problem, which is that the identifiers in Scheme are not always legal identifiers in C. The function `IdScheme->IdC` mangles legal Scheme identifiers into legal C identifiers. In doing so, the main problem is to eliminate illegal characters while still insuring a translation plan that keeps the name more or less reversible so that when we encounter an identifier in C we can reconstruct the original name of the variable in Scheme. There are many solutions to this problem, but ours is to translate problematic characters into normal ones and then to verify whether the name we get that way has already been used so that we avoid name conflicts. The code for that is not particularly interesting but, just so we hide nothing, here are the details of those functions. The variable `Scheme->C-names-mapping` stores all the translations of names. It is predefined with a few translations already imposed on it. Others can be added there, though we've omitted them here.

```
(define Scheme->C-names-mapping
  '( (*         . "TIMES")
     (<         . "LESSP")
     (pair?     . "CONSP")
     (set-cdr! . "RPLACD")
     ) )
(define (IdScheme->IdC name)
  (let ((v (assq name Scheme->C-names-mapping)))
    (if (pair? v) (cdr v)
        (let ((str (symbol->string name)))
          (let retry ((Cname (compute-Cname str)))
            (if (Cname-clash? Cname Scheme->C-names-mapping)
                (retry (compute-another-Cname str))
                (begin (set! Scheme->C-names-mapping
                             (cons (cons name Cname)
                                   Scheme->C-names-mapping ) )
                       Cname ) ) ) ) ) ) )
```

When there is a conflict, we synthesize a new name by suffixing the original name by an increasing index.

```
(define (Cname-clash? Cname mapping)
  (let check ((mapping mapping))
    (and (pair? mapping)
         (or (string=? Cname (cdr (car mapping)))
             (check (cdr mapping)) ) ) ) )
(define compute-another-Cname
  (let ((counter 1))
    (lambda (str)
      (set! counter (+ 1 counter))
```

```
      (compute-Cname (format #f "~A_~A" str counter)) ) ) )
(define (compute-Cname str)
  (define (mapcan f l)
    (if (pair? l)
        (append (f (car l)) (mapcan f (cdr l)))
        '() ) )
  (define (convert-char char)
    (case char
      ((#\_)              '(#\_ #\_))
      ((#\?)              '(#\p))
      ((#\!)              '(#\i))
      ((#\<)              '(#\l))
      ((#\>)              '(#\g))
      ((#\=)              '(#\e))
      ((#\- #\/ #\* #\:) '())
      (else               (list char)) ) )
  (let ((cname (mapcan convert-char (string->list str))))
    (if (pair? cname) (list->string cname) "weird") ) )
```

An isolated underscore cannot appear in the generated names. We'll use that character, for example, in the names of temporary variables, with no risk of confusion. This translation plan cannot translate the name `1+`, but since that one is not obligatorily a legal identifier in Scheme, we're not going to worry too long about it.

10.8.2 Quotations

Quotations have to be translated into a fragment of C that can be retrieved at execution time. We'll introduce an innovation here by presenting a translation plan where they will be translated only by declarations of data, excluding all executable code. We also guarantee that quotations will take the least possible space. That is, subexpressions will be identified so they can be shared.

The function `generate-quotations` globally insures the translation of quotations. To simplify what follows, we'll ignore the case of vectors, large integers, rationals, floating-point numbers, and any characters that can be handled without additional considerations.

```
(define (generate-quotations out qv*)
  (when (pair? qv*)
    (format out "~%/* Quotations: */~%")
    (scan-quotations out qv* (length qv*) '()) ) )
```

What's actually happening is that `scan-quotations` analyzes the quoted values as they appear in the instances of `Quotation-Variable`. Whenever possible, these values are shared. The predicate `already-seen-value?` detects that possibility.

```
(define (scan-quotations out qv* i results)
  (when (pair? qv*)
    (let* ((qv       (car qv*))
           (value    (Quotation-Variable-value qv))
           (other-qv (already-seen-value? value results)) )
      (cond (other-qv
```

```
            (generate-quotation-alias out qv other-qv)
            (scan-quotations out (cdr qv*) i (cons qv results)) )
           ((C-value? value)
            (generate-C-value out qv)
            (scan-quotations out (cdr qv*) i (cons qv results)) )
           ((symbol? value)
            (scan-symbol out value qv* i results) )
           ((pair? value)
            (scan-pair out value qv* i results) )
           (else (generate-error "Unhandled constant" qv)) ) ) ) )
(define (already-seen-value? value qv*)
  (and (pair? qv*)
       (if (equal? value (Quotation-Variable-value (car qv*)))
           (car qv*)
           (already-seen-value? value (cdr qv*)) ) ) )
```

All the quotations are identified in C by lexemes prefixed by **thing**. Detecting something that can be shared is a matter of simply identifying two lexemes themselves. We do that with a C macro that implements **generate-quotation-alias**. To make it easier to read the generated program, we comment the shared value.

```
(define (generate-quotation-alias out qv1 qv2)
  (format out "#define thing~A thing~A /* ~S */~%"
          (Quotation-Variable-name qv1)
          (Quotation-Variable-name qv2)
          (Quotation-Variable-value qv2) ) )
```

The predicate **C-value?** tests whether values are immediate. When they are, we translate them into C with no further ado. The function **generate-C-value** actually does that. Immediate values are the empty list, Booleans, short integers, and character strings. All those values are translated into appropriate C objects, and we'll assume for the moment that there are appropriate C macros to define character strings, integers, both Booleans, and the empty list. All these C entities are prefixed by **SCM_**. No doubt C gurus will notice that the values restricting small numbers limit them to 16-bits in one's complement.

```
(define *maximal-fixnum* 16384)
(define *minimal-fixnum* (- *maximal-fixnum*))
(define (C-value? value)
  (or (null? value)
      (boolean? value)
      (and (integer? value)
           (< *minimal-fixnum* value)
           (< value *maximal-fixnum*) )
      (string? value) ) )
(define (generate-C-value out qv)
  (let ((value (Quotation-Variable-value qv))
        (index (Quotation-Variable-name qv)) )
    (cond ((null? value)
           (format out "#define thing~A SCM_nil /* () */~%"
                   index ) )
          ((boolean? value)
```

```
        (format out "#define thing~A ~A /* ~S */~%"
                index (if value "SCM_true" "SCM_false") value ) )
       ((integer? value)
        (format out "#define thing~A SCM_Int2fixnum(~A)~%"
                index value ) )
       ((string? value)
        (format out "SCM_DefineString(thing~A_object,\"~A\");~%"
                index value )
        (format out "#define thing~A SCM_Wrap(&thing~A_object)~%"
                index index ) ) ) ) )
```

When values are composites (that is, dotted pairs or symbols), we decompose them to determine whether sharing is possible so that we can rebuild them later from their components. A symbol is reconstructed only from the characters in its name. Strings are created prior to symbols.

```
(define (scan-symbol out value qv* i results)
  (let* ((qv    (car qv*))
         (str   (symbol->string value))
         (strqv (already-seen-value? str results)) )
    (cond (strqv (generate-symbol out qv strqv)
                 (scan-quotations out (cdr qv*) i (cons qv results)) )
          (else
           (let ((newqv (make-Quotation-Variable
                         i (symbol->string value) )))
             (scan-quotations out (cons newqv qv*)
                              (+ i 1) results ) ) ) ) ) )
(define (generate-symbol out qv strqv)
  (format out "SCM_DefineSymbol(thing~A_object,thing~A);    /* ~S */~%"
          (Quotation-Variable-name qv)
          (Quotation-Variable-name strqv)
          (Quotation-Variable-value qv) )
  (format out "#define thing~A SCM_Wrap(&thing~A_object)~%"
          (Quotation-Variable-name qv) (Quotation-Variable-name qv) ) )
```

For dotted pairs, we begin by generating their `car` and then their `cdr`, searching for any possible shared things. Sharing might even occur between the `car` and `cdr`. The programming style by continuations inspires the function `scan-pair`, as you can see.

```
(define (scan-pair out value qv* i results)
  (let* ((qv  (car qv*))
         (d   (cdr value))
         (dqv (already-seen-value? d results)) )
    (if dqv
        (let* ((a   (car value))
               (aqv (already-seen-value? a results)) )
          (if aqv
              (begin
                (generate-pair out qv aqv dqv)
                (scan-quotations out (cdr qv*) i (cons qv results)) )
              (let ((newaqv (make-Quotation-Variable i a)))
                (scan-quotations out (cons newaqv qv*)
```

```
                                (+ i 1) results ) ) ) )
        (let ((newdqv (make-Quotation-Variable i d)))
          (scan-quotations
           out (cons newdqv qv*) (+ i 1) results ) ) ) ) )
(define (generate-pair out qv aqv dqv)
  (format out
          "SCM_DefinePair(thing~A_object,thing~A,thing~A); /* ~S */~%"
          (Quotation-Variable-name qv)
          (Quotation-Variable-name aqv)
          (Quotation-Variable-name dqv)
          (Quotation-Variable-value qv) )
  (format out "#define thing~A SCM_Wrap(&thing~A_object)~%"
          (Quotation-Variable-name qv) (Quotation-Variable-name qv) ) )
```

Now we're going to look at an example and get into the details of data representation.

10.8.3 Declaring Data

Let's take a simple program made of a single quotation: `(quote ((#F #T) (FOO . "FOO") 33 FOO . "FOO"))`. The sort of compilation that we have just described leads to what follows here. There are a few comments to elucidate it. Enjoy!

```
/* Source expression:
'((#F #T) (FOO . "FOO") 33 FOO . "FOO")  */

#include "scheme.h"

/* Quotations */
SCM_DefineString(thing4_object,"FOO");
#define thing4 SCM_Wrap(&thing4_object)
SCM_DefineSymbol(thing5_object,thing4);        /* FOO */
#define thing5 SCM_Wrap(&thing5_object)
SCM_DefinePair(thing3_object,thing5,thing4);   /* (FOO . "FOO") */
#define thing3 SCM_Wrap(&thing3_object)
#define thing6 SCM_Int2fixnum(33)
SCM_DefinePair(thing2_object,thing6,thing3);   /* (33 FOO . "FOO") */
#define thing2 SCM_Wrap(&thing2_object)
SCM_DefinePair(thing1_object,thing3,thing2);
                                /* ((FOO . "FOO") 33 FOO . "FOO") */
#define thing1 SCM_Wrap(&thing1_object)
#define thing9 SCM_nil                         /* () */
#define thing10 SCM_true                       /* #T */
SCM_DefinePair(thing8_object,thing10,thing9);  /* (#T) */
#define thing8 SCM_Wrap(&thing8_object)
#define thing11 SCM_false                      /* #F */
SCM_DefinePair(thing7_object,thing11,thing8);  /* (#F #T) */
#define thing7 SCM_Wrap(&thing7_object)
SCM_DefinePair(thing0_object,thing7,thing1);
                        /* ((#F #T) (FOO . "FOO") 33 FOO . "FOO") */
```

```
#define thing0 SCM_Wrap(&thing0_object)
                    ...
```

The first thing we create is the character string "FOO". To do that, we use the macro SCM_DefineString. As its first argument, it takes the name the object will have in C. As its second argument, it takes the character string. Similarly, a symbol is created by the macro SCM_DefineSymbol. As its first argument, it also takes the name that the object will have in C, and as its second argument, it takes the character string that names this symbol. Likewise, a dotted pair is created by the macro SCM_DefinePair. As its first argument, it, too, takes the name the object will have in C, and its second and third arguments are the contents of the car and cdr.

Predefined objects like Booleans or the empty list, of course, are not created anew. Rather, they are referred to by the names SCM_true, SCM_false, and SCM_nil.

Later, [see p. 390], we'll indicate the exact representations of values in Scheme. Compilation is largely independent of the particular representation that we adopt. For the time being, it suffices to know that the creation directives like SCM_Define... allocate only objects and that we get the legal values referring to them by converting their address with SCM_Wrap. As for short integers, they are converted by SCM_Int2fixnum. For that reason, after any object is allocated, we define the C value representing it under a name beginning with thing. For example, thing4 indicates the character string "FOO" while thing6 is the short integer 33. More precisely, thing4 is the pointer to the object thing4_object, which (strictly speaking) is the character string.

The character string "FOO" is shared, as is the S-expression (FOO . "FOO"), between the objects thing4 and thing3.

10.8.4 Compiling Expressions

Compiling expressions from Scheme into C is, of course, the main task of our compiler. Here again we are innovating because we translate expressions into expressions. That way, we gain a certain clarity in the result, which respects the structure of the source program. In spite of its obvious wisdom, this choice nevertheless poses a delicate problem since it deviates slightly from the philosophy of C. C prefers instructions to expressions. This deviation is not too troublesome except when we are debugging, say, with gdb, the symbolic debugger from the *Free Software Foundation*. In single-step mode, execution occurs instruction by instruction—too gross a granularity for our choice. If we had wanted to compile into instructions (rather than into expressions), we would have adopted a technique similar to that the byte-code compiler. [see p. 223]

The actual compilation of expressions is carried out by a generic function, ->C. Its first argument is the expression to compile, of course, and its second argument is the output port on which to write the resulting C. Since we are writing to a file, we have to pay close attention to producing the code sequentially, without backtracking, in a single pass. [see p. 234]

```
(define-generic (->C (e Program) out))
```

In contrast to the preceding transformations, we're not going to use our generic code walker (because there is no default treatment here), but rather a method of generating code for each type of syntactic node. Thus there is nothing left for us to do except to enumerate the methods. They are simple enough since they systematically copy the equivalent C constructions.

As a language, C embodies very strong ideas of syntax with its precedence, variable meanings for associativity, and so forth. Since the practice is widely recommended, we are going to use parentheses throughout. For Lisp fans, this practice will add a Lisp-like flavor, though it may be distasteful for habitual C users. The parentheses will be specified by the macro **between-parentheses**.

```
(define-syntax between-parentheses
  (syntax-rules ()
    ((between-parentheses out . body)
     (let ((out out))
       (format out "(")
       (begin . body)
       (format out ")") ) ) ) )
```

Compiling References to Variables

There's more than one type of variable that we have to handle, but in general, we assimilate a Scheme variable with a C variable. The appropriate method will be subcontracted to the function **reference->C**, which will generally subcontract it immediately to the function **variable->C**. These indirections enable us to specialize their behavior more easily.

```
(define-method (->C (e Reference) out)
  (reference->C (Reference-variable e) out) )
(define-generic (reference->C (v Variable) out))
(define-method (reference->C (v Variable) out)
  (variable->C v out) )
(define-generic (variable->C (variable) out))
```

In a general way, a variable is translated by name into C, except when it has been renamed or when it indicates a quotation.

```
(define-method (variable->C (variable Variable) out)
  (format out (IdScheme->IdC (Variable-name variable))) )
(define-method (variable->C (variable Renamed-Local-Variable) out)
  (format out "~A_~A"
          (IdScheme->IdC (Variable-name variable))
          (Renamed-Local-Variable-index variable) ) )
(define-method (variable->C (variable Quotation-Variable) out)
  (format out "thing~A" (Quotation-Variable-name variable)) )
```

However, there's a particular case for non-predefined global variables—the free variables in the compiled program—because no analysis has told us whether or not they have been initialized. We thus must verify that fact explicitly by means of the macro **SCM_CheckedGlobal**. [see **Ex. 10.2**]

```
(define-method (reference->C (v Global-Variable) out)
  (format out "SCM_CheckedGlobal")
```

```
    (between-parentheses out
      (variable->C v out) ) )
```

The remaining case is that of free variables for which we will once more call the appropriate macro: `SCM_Free`.

```
(define-method (->C (e Free-Reference) out)
  (format out "SCM_Free")
  (between-parentheses out
    (variable->C (Free-Reference-variable e) out) ) )
```

Compiling Assignments

Assignments are handled even more simply because there are only two kinds: assignments of global variables and assignments written in boxes. The assignment of a global variable is translated by the assignment in C of the corresponding global variable.

```
(define-method (->C (e Global-Assignment) out)
  (between-parentheses out
    (variable->C (Global-Assignment-variable e) out)
    (format out "=")
    (->C (Global-Assignment-form e) out) ) )
```

Compiling Boxes

As for boxes, there are only three operations involved. A C macro read- or write-accesses the contents of a box: `SCM_Content`. A function of the predefined library, `SCM_allocate_box`, allocates a box when needed.

```
(define-method (->C (e Box-Read) out)
  (format out "SCM_Content")
  (between-parentheses out
    (->C (Box-Read-reference e) out) ) )
(define-method (->C (e Box-Write) out)
  (between-parentheses out
    (format out "SCM_Content")
    (between-parentheses out
      (->C (Box-Write-reference e) out) )
    (format out "=")
    (->C (Box-Write-form e) out) ) )
(define-method (->C (e Box-Creation) out)
  (variable->C (Box-Creation-variable e) out)
  (format out "= SCM_allocate_box")
  (between-parentheses out
    (variable->C (Box-Creation-variable e) out) ) )
```

Compiling Alternatives

The Scheme alternative is translated into the alternative expression in C, that is, the ternary construction $\pi_0?\pi_1:\pi_2$. Since every value different from False is con-

sidered True in Scheme, we explicitly test this case. Of course, we put parentheses in everywhere!

```
(define-method (->C (e Alternative) out)
  (between-parentheses out
    (boolean->C (Alternative-condition e) out)
    (format out "~%? ")
    (->C (Alternative-consequent e) out)
    (format out "~%: ")
    (->C (Alternative-alternant e) out) ) )
(define-generic (boolean->C (e) out)
  (between-parentheses out
    (->C e out)
    (format out " != SCM_false") ) )
```

There you see one of the first sources of inefficiency in comparison to a real compiler. In the case of a predicate like `(if (pair? x) ... )`, it is inefficient for the call to `pair?` to return a Scheme Boolean that we compare to `SCM_false`. It would be a better idea for `pair?` to return a C Boolean directly. We could thus specialize the function `boolean->C` to recognize and handle calls to predefined predicates. We could also unify the Booleans of C and Scheme by adopting the convention that False is represented by `NULL`. Then we could carry out type recovery in order to suppress these cumbersome comparisons, as in [Shi91, Ser93, WC94].

Compiling Sequences

Sequences are translated into C sequences in this notation $(\pi_1, \ldots, \pi_n)$.

```
(define-method (->C (e Sequence) out)
  (between-parentheses out
    (->C (Sequence-first e) out)
    (format out ",~%")
    (->C (Sequence-last e) out) ) )
```

10.8.5 Compiling Functional Applications

We've organized functional applications into several categories: normal functional applications, closed functional applications (where the function term is an abstraction), functional applications where the invoked function is a function value of a predefined variable. These three types of applications are produced by three classes of syntactic nodes: `Regular-Application`, `Fix-Let`, and `Predefined-Application`.

A normal functional application $(f\ x_1\ \ldots\ x_n)$ will be compiled into a C expression `SCM_invoke(`f`,`n`,`x_1`,... ,`x_n`)`, where n is the number of arguments passed to the function and `SCM_invoke` is the specialized invocation function. To make the result more legible, we'll use C macros for arity of less than four, like this:

```
#define SCM_invoke0(f)       SCM_invoke(f,0)
#define SCM_invoke1(f,x)     SCM_invoke(f,1,x)
#define SCM_invoke2(f,x,y)   SCM_invoke(f,2,x,y)
#define SCM_invoke3(f,x,y,z) SCM_invoke(f,3,x,y,z)
```

For greater arity, we'll use this protocol directly:

```
(define-method (->C (e Regular-Application) out)
  (let ((n (number-of (Regular-Application-arguments e))))
    (cond ((< n 4)
           (format out "SCM_invoke~A" n)
           (between-parentheses out
             (->C (Regular-Application-function e) out)
             (->C (Regular-Application-arguments e) out) ) )
          (else (format out "SCM_invoke")
                (between-parentheses out
                  (->C (Regular-Application-function e) out)
                  (format out ",~A" n)
                  (->C (Regular-Application-arguments e) out) ) ) ) ) )
```

Closed applications will be translated into a C sequence corresponding to their body. *A posteriori*, they will justify our effort in collecting temporary variables beforehand. By the way, that collecting is equivalent to *frame coalescing*, or fusing temporary blocks, a compilation technique where we attempt to limit the number of allocations by allocating larger blocks, as in [AS94, PJ87]. If we assume that all temporary variables in the `let` forms have already been allocated, then binding these variables to their values is just an assignment. We could translate that assignment, in Lispian terms, by a rewrite rule like the following[4] where we coalesce the internal `let`s into a unique `let` surrounding all of them and associated with the local renamings. That transformation is not really legal in the general case because of this possibility: if $\ldots_1$ returns more than once, and if π_2 captures the variable `x`, then this `x` renamed as `x1` will be shared. [see p. 189] Nevertheless, the transformation is correct here because `x1` will not be modified; the variable is immutable, even if `set!` is present since it has been boxed; and we're using a technique of a flat environment where values are recopied.

```
(begin ...1                          (let (x1 x2) ...1
  (let ((x π1))                        (set! x1 π1)
    π2 )                               π2[x→x1]
  ...2                          ⇒      ...2
  (let ((x π3))                        (set! x2 π3)
    π4 ) )                             π4[x→x2] )
```

We only have to assign each local variable its initialization value. Since all the local variables have been identified and renamed, there's no possible confusion, nor any scope problems. A particular generic function, `bindings->C`, translates those bindings.

```
(define-method (->C (e Fix-Let) out)
  (between-parentheses out
    (bindings->C (Fix-Let-variables e) (Fix-Let-arguments e) out)
    (->C (Fix-Let-body e) out) ) )
(define-generic (bindings->C variables (arguments) out))
(define-method (bindings->C variables (e Arguments) out)
  (variable->C (car variables) out)
```

4. Of course, we could try to share even more of the places reserved for variables. That's the case for the variables `x1` and `x2` if there is no risk of confusion.

```
    (format out "=")
    (->C (Arguments-first e) out)
    (format out ",~%")
    (bindings->C (cdr variables) (Arguments-others e) out) )
  (define-method (bindings->C variables (e No-Argument) out)
    (format out "") )
```

Finally, the case of functional applications where the function is the value of a predefined variable and thus well known: they will be *inlined*. The function will be called directly without the intercession of `SCM_invoke`. Predefined functions are written in C and appear in the library that must be linked to the compiled program. Calling them directly is equivalent but of course more efficient than passing through `SCM_invoke`.

Since generating the direct call depends on how the primitive is implemented, we assume that the functional description associated with the predefined variable has a field—`generator`—that indicates the right generator to call. That generation function will receive the node corresponding to the application and will be responsible for calling `->C` recursively on the arguments. It uses the generic function `arguments->C` to make those recursive calls.

```
(define-method (->C (e Predefined-Application) out)
  ((Functional-Description-generator
    (Predefined-Variable-description
     (Predefined-Application-variable e) ) ) e out ) )
(define-generic (arguments->C (e) out))
(define-method (arguments->C (e Arguments) out)
  (->C (Arguments-first e) out)
  (->C (Arguments-others e) out) )
(define-method (arguments->C (e No-Argument) out)
  #t )
```

10.8.6 Predefined Environment

When we compile applications into predefined functions, the compiler must already know those functions, so we will define a macro, `defprimitive`, for that purpose.

```
(define-class Functional-Description Object (comparator arity generator))

(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name Cname arity)
     (let ((v (make-Predefined-Variable
               'name (make-Functional-Description
                      = arity
                      (make-predefined-application-generator 'Cname) ) )))
       (set! g.init (cons v g.init))
       'name ) ) ) )
```

Thus the definition of a primitive function is based on its name in Scheme, its name in C, and its arity. The generated calls all take the form of a function call in C, that is, a name followed by a mass of arguments within parentheses separated

by commas. The function **make-predefined-application-generator** creates just such generators.

```
(define (make-predefined-application-generator Cname)
  (lambda (e out)
    (format out "~A" Cname)
    (between-parentheses out
      (arguments->C (Predefined-Application-arguments e) out) ) ) )
```

Let's look at a few examples, like the usual **cons**, **car**, +, or =.

```
(defprimitive cons "SCM_cons" 2)
(defprimitive car "SCM_car" 1)
(defprimitive + "SCM_Plus" 2)
(defprimitive = "SCM_EqnP" 2)
```

You see that **cons** is compiled into a call to the C function **SCM_cons** whereas + is compiled into a call to the C macro **SCM_Plus**. [see p. 394] Macros are distinguished from functions by capital letters in the names of the macros. Distinguishing them this way changes the speed at execution and the size of the resulting C code.

10.8.7 Compiling Functions

The preceding transformations have replaced abstractions by allocations of closures that is, nodes from the class **Closure-Creation**. These closures are all allocated by the function **SCM_close**, part of the predefined execution library. As its first argument, this function takes the address of the C function (conveniently typed by the C macro **SCM_CfunctionAddress**) corresponding to the body of the abstraction; the arity of the closure being created is its second argument; those two arguments are followed by the number of closed values, prefixing these same values. We get all those closed values simply by translating them into C from the **free** field of the description of the closure. The arity of functions that take a fixed number of arguments will be coded by that same number. In contrast, when a function has a dotted variable, it must take at least a certain number of arguments, say, i, so its arity will thus be represented by $-i-1$. That is, the function **list** has an arity of -1.

```
(define-method (->C (e Closure-Creation) out)
  (format out "SCM_close")
  (between-parentheses out
    (format out "SCM_CfunctionAddress(function_~A),~A,~A"
            (Closure-Creation-index e)
            (generate-arity (Closure-Creation-variables e))
            (number-of (Closure-Creation-free e)) )
    (->C (Closure-Creation-free e) out) ) )
(define (generate-arity variables)
  (let count ((variables variables)(arity 0))
    (if (pair? variables)
        (if (Local-Variable-dotted? (car variables))
            (- (+ arity 1))
            (count (cdr variables) (+ 1 arity)) )
        arity ) ) )
```

```
(define-method (->C (e No-Free) out)
  #t )
(define-method (->C (e Free-Environment) out)
  (format out ",")
  (->C (Free-environment-first e) out)
  (->C (Free-environment-others e) out) )
```

Abstractions themselves have been organized into the object representing the entire program, an instance of **Flattened-Program**, as a list of functions without free variables defined by the instances of **With-Temp-Function-Definition**. Each of those functions leads to generating an equivalent function in C.

```
(define (generate-functions out definitions)
  (format out "~%/* Functions: */~%")
  (for-each (lambda (def)
              (generate-closure-structure out def)
              (generate-possibly-dotted-definition out def) )
            (reverse definitions) ) )
```

To make the generated code more legible, we will again use C macros to hide the finer details of representation. A function will be generated for each closure, along with a data structure defining where the enclosed variables are located in the object of type closure. All the names generated to represent these objects will be formed from the root `function_` and an index that already appears in the **index** field of the object **Function-Definition**.[5]

The representation of the closure is defined by the macro `SCM_DefineClosure`. Its first argument is the name of the associated C function; its second argument is the list of names of captured variables, separated by semi-colons.

```
(define (generate-closure-structure out definition)
  (format out "SCM_DefineClosure(function_~A, "
          (Function-Definition-index definition) )
  (generate-local-temporaries (Function-Definition-free definition) out)
  (format out ");~%") )
```

The function **generate-possibly-dotted-definition** generates the definition of the C function by taking into account its arity. The function is defined by the macro `SCM_DeclareFunction`. It takes the name of the generated function as an argument. Its variables are defined by the macros `SCM_DeclareLocalVariable` or `SCM_DeclareLocalDottedVariable`. They take the name of the variable and its rank in the list of variables. The rank is important only for computing the list bound to a dotted variable. The body of the function does not pose a problem. We get it simply by applying the function **->C**, but that is preceded by a **return**, necessary to the C functions.

```
(define (generate-possibly-dotted-definition out definition)
  (format out "~%SCM_DeclareFunction(function_~A) {~%"
          (Function-Definition-index definition) )
  (let ((vars (Function-Definition-variables definition))
        (rank -1) )
```

5. Since the functions are named like this: `function`i, you can see that they are independent of the variables in which they are stored. Of course, it would be better if the functions associated with these non-mutable global variables were named after them, too.

```
(for-each (lambda (v)
            (set! rank (+ rank 1))
            (cond ((Local-Variable-dotted? v)
                   (format out "SCM_DeclareLocalDottedVariable(") )
                  ((Variable? v)
                   (format out "SCM_DeclareLocalVariable(") ) )
            (variable->C v out)
            (format out ",~A);~%" rank) )
          vars )
(let ((temps (With-Temp-Function-Definition-temporaries
               definition )))
  (when (pair? temps)
    (generate-local-temporaries temps out)
    (format out "~%") ) )
(format out "return ")
(->C (Function-Definition-body definition) out)
(format out ";~%}~%~%") ) )
```

Lists of variables are converted into C by means of the utility function `generate-local-temporaries`.

```
(define (generate-local-temporaries temps out)
  (when (pair? temps)
    (format out "SCM ")
    (variable->C (car temps) out)
    (format out "; ")
    (generate-local-temporaries (cdr temps) out) ) )
```

10.8.8 Initializing the Program

We have an entire Scheme program to compile now, so the only thing left for us to indicate is how to form an entire C program. For that reason, we put the initial expression into a closure during the phase of flattening out the functions. [see p. 369] The only major and arguable decision is that we generate a call to the function `SCM_print`, which will print the value of the compiled expression. It's not strictly necessary, and there are many interesting programs that do useful things without polluting the world with their output (`cc`, for example). For us, though, it is more convenient for the little programs that we compile to have printed output.

```
(define (generate-main out form)
  (format out "~%/* Expression: */~%void main(void) {~%")
  (format out "  SCM_print")
  (between-parentheses out
    (->C form out) )
  (format out ";~%  exit(0);~%}~%") )
```

Of course, we haven't yet said anything about the representation of data, nor have we defined the initial execution library. However, the compiler is complete and operational anyway, so we are going to translate our current example into C. (We manually indented the generated output to make it more legible.) Here, then, is the complete translation of our example, which by the way appears as a comment from lines 2 to 9.

[see p. 360]

o/chap10ex.c

```
/* Compiler to C $Revision: 4.0 $
(BEGIN
  (SET! INDEX 1)
  ((LAMBDA
     (CNTER . TMP)
     (SET! TMP (CNTER (LAMBDA (I) (LAMBDA X (CONS I X)))))
     (IF CNTER (CNTER TMP) INDEX))
    (LAMBDA (F) (SET! INDEX (+ 1 INDEX)) (F INDEX))
    'FOO))  */

#include "scheme.h"

/* Global environment: */
SCM_DefineGlobalVariable(INDEX,"INDEX");

/* Quotations: */
#define thing3 SCM_nil /* () */
SCM_DefineString(thing4_object,"FOO");
#define thing4 SCM_Wrap(&thing4_object)
SCM_DefineSymbol(thing2_object,thing4);       /* FOO */
#define thing2 SCM_Wrap(&thing2_object)
#define thing1 SCM_Int2fixnum(1)
#define thing0 thing1 /* 1 */

/* Functions: */
SCM_DefineClosure(function_0, );

SCM_DeclareFunction(function_0) {
  SCM_DeclareLocalVariable(F,0);
  return ((INDEX=SCM_Plus(thing1,
                          SCM_CheckedGlobal(INDEX))),
          SCM_invoke1(F,
                      SCM_CheckedGlobal(INDEX)));
}

SCM_DefineClosure(function_1, SCM I; );

SCM_DeclareFunction(function_1) {
  SCM_DeclareLocalDottedVariable(X,0);
  return SCM_cons(SCM_Free(I),
                  X);
}

SCM_DefineClosure(function_2, );

SCM_DeclareFunction(function_2) {
```

```
48   SCM_DeclareLocalVariable(I,0);
49   return SCM_close(SCM_CfunctionAddress(function_1),-1,1,I);
50 }
51
52 SCM_DefineClosure(function_3, );
53
54 SCM_DeclareFunction(function_3) {
55   SCM TMP_2; SCM CNTER_1;
56   return ((INDEX=thing0),
57           (CNTER_1=SCM_close(SCM_CfunctionAddress(function_0),1,0),
58            TMP_2=SCM_cons(thing2,
59                           thing3),
60            (TMP_2= SCM_allocate_box(TMP_2),
61             ((SCM_Content(TMP_2)=
62              SCM_invoke1(CNTER_1,SCM_close(SCM_CfunctionAddress
63                                            (function_2),1,0))),
64              ((CNTER_1 != SCM_false)
65               ? SCM_invoke1(CNTER_1,
66                             SCM_Content(TMP_2))
67               : SCM_CheckedGlobal(INDEX))))));
68 }
69
70
71 /* Expression: */
72 void main(void) {
73   SCM_print(SCM_invoke0(SCM_close(SCM_CfunctionAddress(function_3),
74                                   0,0)));
75   exit(0);
76 }
77
78 /* End of generated code. */
79
```

We'll also give its expanded form, that is, one stripped of the C macros (except for `va_list`), for the functions `function_1` and `function_2`. The other expansions are from the same tap.

```
struct function_1 {
        SCM(*behavior) (void);
        long            arity;
        SCM             I;
};

SCM function_1(struct function_1 * self_,
           unsigned long size_,
           va_list arguments_)
{
        SCM             X = SCM_list(size_ - 0, arguments_);
        return SCM_cons(((*self_).I), X);
}
```

```
struct function_2 {
        SCM(*behavior) (void);
        long            arity;
};

SCM function_2(struct function_2 * self_,
           unsigned long size_,
           va_list arguments_)
{
        SCM             I = va_arg(arguments_, SCM);
        return SCM_close(((SCM(*) (void)) function_1), -1, 1, I);
}
```

If we compile it with a compiler conforming to ISO-C (one like `gcc` for example) with appropriate execution libraries like we've already described, then we get (a little tiny) executable that (very rapidly[6]) produces the value[7] we want.

```
% gcc -ansi -pedantic chap10ex.c scheme.o schemelib.o
% time a.out
(2 3)
0.010u 0.000s 0:00.00 0.0% 3+5k 0+0io 0pf+0w
% size a.out
text    data    bss     dec     hex
28672   4096    32      32800   8020
```

10.9 Representing Data

Now we're going to clarify the set of C macros in the file **`scheme.h`**. No need to repeat that the point of this exercise is not to deliver the most high-performance compiler possible, but rather to outline, explain, and demonstrate various techniques! We won't burden ourselves with problems like memory management; there is no garbage collector, and all allocations use the function **`malloc`**, making the adaptation of a conservative garbage collector—like the one developed by Hans Boehm in [BW88]—particularly simple.

Values in Lisp and Scheme are such that we can always inquire about their type. Consequently, it is necessary for that information about types to be associated with each value. Making the cost of this association as low as possible is the source of much torment for implementers. Values are also of variable size since they can contain an arbitrary number of values themselves. The work-around is to manipulate these values by their address, since an address has a fixed size. Objects will be allocated in memory, and their type will be encoded as the word preceding them. The inconvenience (in comparison to a statically typed language where types no longer even exist by execution time) is that simple values (like short integers might be) can no longer be handled directly since we have to follow a pointer to get one of them. There are many solutions to this problem. On many machines,

6. The time that appears here corresponds mostly to the time for loading the program; the part corresponding to execution is negligible. Later, you'll see how iterating the computation 10000 times gives a time of about 1 second.

7. The benchmarks were measured on a Sony News 3200 with a Mips R3000 processor.

addresses are multiples of four, leaving the two least significant bits free in an address. We can appropriate those bits, for example, to encode the kind of object being pointed to. And if the referenced value is an integer, then why not put it in place of the address? That's what we'll do for integers, making them more efficient but stripping off a bit and thus limiting their range. Other schemes have also been invented, and they are catalogued in [Gud93].

To get back to the task at hand, look at Figure 10.5. A value in Scheme will be represented by a value of C such that if its least significant bit is 1, then it is an integer coded in the remaining bits. We'll be talking about 31-bit integers for a machine of 32-bit words. If the least significant bit is 0, then we're dealing with the address of the first field of an allocated value. The type of that value appears in the word that precedes this address[8] as in [DPS94a]. Such a value appears as a real pointer in C, referring directly to the interesting values of the object. The type of Scheme values in C—what we'll be handling all the time—will be called `SCM`:

```
typedef union SCM_object          *SCM;
```

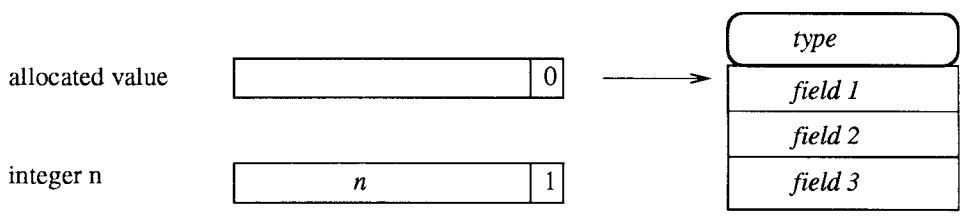


Figure 10.5 Representation of Scheme values in C

The macro `SCM_FixnumP` distinguishes a short integer from a pointer. Small integers and values can be converted back and forth by means of `SCM_Fixnum2int` and `SCM_Int2fixnum`. The integer 37 in Scheme is thus represented by the integer 75 in C.

```
#define SCM_FixnumP(x)     ((unsigned long)(x) & (unsigned long)1)
#define SCM_Fixnum2int(x) ((long)(x)>>1)
#define SCM_Int2fixnum(i) ((SCM)(((i)<<1) | 1))
```

When a value of `SCM` is not a short integer, it's a pointer to an object defined by `SCM_object`. That's a union of various possible types. Some types are missing, like floating-point numbers, vectors, input and output ports. However, you will find the essential types there: dotted pairs (with `cdr` in the lead to remind you that you're dealing with a list and also because `cdr` is accessed more frequently than `car`, according to [Cla79]).

8. The null pointer for C, `NULL`, is generally implemented as `0L`. It is not a legal pointer for our compiler because there is seldom a word at the address -1.

```
union SCM_object {
  struct SCM_pair {
    SCM cdr;
    SCM car;
  } pair;
  struct SCM_string {
    char Cstring[8];
  } string;
  struct SCM_symbol {
    SCM pname;
  } symbol;
  struct SCM_box {
    SCM content;
  } box;
  struct SCM_subr {
    SCM (*behavior)();
    long arity;
  } subr;
  struct SCM_closure {
    SCM (*behavior)();
    long arity;
    SCM environment[1];
  } closure;
  struct SCM_escape {
    struct SCM_jmp_buf *stack_address;
  } escape;
};
```

First class objects among the variations of `SCM_object` are prefixed by their type. You can't see the type until after a little C magic. The type will be represented by an explicitly enumerated type: `SCM_tag`. That leaves the number of bits free for other purposes, like for use by the garbage collector. The type will be stored in a field of type `SCM_header`, which will be the same size as an `SCM`, justifying the data member `SCM ignored` in the union defining `SCM_header`. Finally, objects prefixed by their type will be defined by the structure `SCM_unwrapped_object`.

```
enum SCM_tag {
  SCM_NULL_TAG          =0xaaa0,
  SCM_PAIR_TAG          =0xaaa1,
  SCM_BOOLEAN_TAG       =0xaaa2,
  SCM_UNDEFINED_TAG     =0xaaa3,
  SCM_SYMBOL_TAG        =0xaaa4,
  SCM_STRING_TAG        =0xaaa5,
  SCM_SUBR_TAG          =0xaaa6,
  SCM_CLOSURE_TAG       =0xaaa7,
  SCM_ESCAPE_TAG        =0xaaa8
};
union SCM_header {
  enum SCM_tag tag;
  SCM ignored;
};
union SCM_unwrapped_object {
  struct SCM_unwrapped_immediate_object {
    union SCM_header header;
  } object;
  struct SCM_unwrapped_pair {
    union SCM_header header;
    SCM cdr;
    SCM car;
  } pair;
  struct SCM_unwrapped_string {
    union SCM_header header;
    char Cstring[8];
  } string;
```

```
  struct SCM_unwrapped_symbol {
    union SCM_header header;
    SCM pname;
  } symbol;
  struct SCM_unwrapped_subr {
    union SCM_header header;
    SCM (*behavior)(void);
    long arity;
  } subr;
  struct SCM_unwrapped_closure {
    union SCM_header header;
    SCM (*behavior)(void);
    long arity;
    SCM environment[1];
  } closure;
  struct SCM_unwrapped_escape {
    union SCM_header header;
    struct SCM_jmp_buf
                *stack_address;
  } escape;
};
```

The following macros convert references back and forth as well as extract its type from an object. In fact, only the execution library needs to distinguish `SCM` from `SCMref`.

```
typedef union SCM_unwrapped_object *SCMref;

#define SCM_Wrap(x)       ((SCM) (((union SCM_header *) x) + 1))
#define SCM_Unwrap(x)     ((SCMref) (((union SCM_header *) x) - 1))
#define SCM_2tag(x)       ((SCM_Unwrap((SCM) x))->object.header.tag)
```

Finally, the addresses of functions written in C (except when applied) are typed as returning an `SCM` and accepting no arguments. The macro `SCM_CfunctionAddress` gets this conversion for us.

```
#define SCM_CfunctionAddress(Cfunction) ((SCM (*)(void)) Cfunction)
```

10.9.1 Declaring Values

There is a set of macros to allocate Scheme values statically. Their names are prefixed by `SCM_Define`. To define a dotted pair, we allocate a structure defined by `SCM_unwrapped_pair`. Defining a symbol is similar. Each time, the object is first created; then a pointer to that object is also created and converted by `SCM_Wrap` into a valid reference.

```
#define SCM_DefinePair(pair,car,cdr) \
  static struct SCM_unwrapped_pair pair = {{SCM_PAIR_TAG}, cdr, car }
#define SCM_DefineSymbol(symbol,pname) \
  static struct SCM_unwrapped_symbol symbol = {{SCM_SYMBOL_TAG}, pname }
```

Defining strings is a little more complicated because C does not know how to initialize data structures of variable size. We also have to be careful in C when we handle strings not to confuse their contents with their address, and we must

not neglect the null character that completes a string. Our solution is to build a definition of an appropriate structure by the defined string. Strings in Scheme will be represented by strings in C without further ado.

```
#define SCM_DefineString(Cname,string) \
  struct Cname##_struct {           \
    union SCM_header header;        \
    char Cstring[1+sizeof(string)];};  \
  static struct Cname##_struct Cname = \
    {{SCM_STRING_TAG}, string }
```

A number of predefined values have to exist. Actually, only their existence and their type (not their contents) are important to us, so they are defined by immediate objects. We call `SCM_Wrap` to convert an address into an `SCM`.

```
#define SCM_DefineImmediateObject(name,tag) \
  struct SCM_unwrapped_immediate_object name = {{tag}}
SCM_DefineImmediateObject(SCM_true_object,SCM_BOOLEAN_TAG);
SCM_DefineImmediateObject(SCM_false_object,SCM_BOOLEAN_TAG);
SCM_DefineImmediateObject(SCM_nil_object,SCM_NULL_TAG);
#define SCM_true        SCM_Wrap(&SCM_true_object)
#define SCM_false       SCM_Wrap(&SCM_false_object)
#define SCM_nil         SCM_Wrap(&SCM_nil_object)
```

Scheme Booleans are not the same as C Booleans, so we must sometimes convert a C Boolean into a Scheme Boolean. `SCM_2bool` does that.

```
#define SCM_2bool(i) ((i) ? SCM_true : SCM_false )
```

We'll also introduce a few supplementary macros to recognize values and and take them apart. The following predicates have names ending with `P` to indicate that they return a C Boolean.

```
#define SCM_Car(x)      (SCM_Unwrap(x)->pair.car)
#define SCM_Cdr(x)      (SCM_Unwrap(x)->pair.cdr)
#define SCM_NullP(x)    ((x)==SCM_nil)
#define SCM_PairP(x)    \
  ((!SCM_FixnumP(x)) && (SCM_2tag(x)==SCM_PAIR_TAG))
#define SCM_SymbolP(x) \
  ((!SCM_FixnumP(x)) && (SCM_2tag(x)==SCM_SYMBOL_TAG))
#define SCM_StringP(x) \
  ((!SCM_FixnumP(x)) && (SCM_2tag(x)==SCM_STRING_TAG))
#define SCM_EqP(x,y)    ((x)==(y))
```

Of course, macros like `SCM_Car` or `SCM_Cdr` will be used in safe situations when we know that the value is a pair. That's not the case for the following arithmetic macros, so they explicitly test whether their arguments are short integers. This style of progamming prevents removing that part of the context where we would know that the arguments are the right type since they always carry out type checking.[9] We're more interested right now in showing all variations for pedagogical reasons. Again, if we were really trying to be efficient, we would organize things quite differently.

9. Or then the C compiler itself takes out these superfluous tests—though that is generally not the case.

```
#define SCM_Plus(x,y)                                          \
 ( ( SCM_FixnumP(x) && SCM_FixnumP(y) )                        \
   ? SCM_Int2fixnum( SCM_Fixnum2int(x) + SCM_Fixnum2int(y) )   \
   : SCM_error(SCM_ERR_PLUS) )
#define SCM_GtP(x,y)                                           \
 ( ( SCM_FixnumP(x) && SCM_FixnumP(y) )                        \
   ? SCM_2bool( SCM_Fixnum2int(x) > SCM_Fixnum2int(y) )        \
   : SCM_error(SCM_ERR_GTP) )
        ...
```

10.9.2 Global Variables

To get the value of a mutable global variable, we use the macro `SCM_CheckedGlobal` to verify whether the variable has been initialized or not. Consequently, we use a C value to indicate that something has not been initialized in Scheme, namely, `SCM_undefined`. Mutable global variables are thus simply initialized (for C) with this value indicating that (for Scheme) they haven't yet been initialized.

```
#define SCM_CheckedGlobal(Cname) \
  ((Cname != SCM_undefined) ? Cname : SCM_error(SCM_ERR_UNINITIALIZED))
#define SCM_DefineInitializedGlobalVariable(Cname,string,value) \
  SCM Cname = SCM_Wrap(value)
#define SCM_DefineGlobalVariable(Cname,string) \
  SCM_DefineInitializedGlobalVariable(Cname,string, \
                                      &SCM_undefined_object)
#define SCM_undefined   SCM_Wrap(&SCM_undefined_object)
SCM_DefineImmediateObject(SCM_undefined_object,SCM_UNDEFINED_TAG);
```

One very important job is to bind predefined values to the global variables by which programs can access them. For example, if we assume that the value of the global variable `NIL` is the empty list, `()`, then we must bind the C variable `NIL` to the C value `SCM_nil_object`. That's the purpose of the file `schemelib.c` containing these definitions among others:

```
SCM_DefineInitializedGlobalVariable(NIL,"NIL",&SCM_nil_object);
SCM_DefineInitializedGlobalVariable(F,"F",&SCM_false_object);
SCM_DefineInitializedGlobalVariable(T,"T",&SCM_true_object);
```

While there is a great deal of folklore surrounding those three variables, `CAR`, `CONS`, and others also have to be available. We get them from the macro `SCM_DefinePredefinedFunctionVariable`. Here is its definition, along with a few examples.

```
#define SCM_DefinePredefinedFunctionVariable(subr,string,\
                                             arity,Cfunction) \
  static struct SCM_unwrapped_subr subr##_object = \
    {{SCM_SUBR_TAG}, Cfunction, arity}; \
  SCM_DefineInitializedGlobalVariable(subr,string,&(subr##_object))
SCM_DefinePredefinedFunctionVariable(CAR,"CAR",1,SCM_car);
SCM_DefinePredefinedFunctionVariable(CONS,"CONS",2,SCM_cons);
SCM_DefinePredefinedFunctionVariable(EQN,"=",2,SCM_eqnp);
SCM_DefinePredefinedFunctionVariable(EQ,"EQ?",2,SCM_eqp);
```

Mutable global variables are put into boxes. Reads and writes go through the macro `SCM_Content`, defined like this:

```
#define SCM_Content(e) ((e)->box.content)
```

10.9.3 Defining Functions

The only thing left to explain is how functions of the program are represented. We have to study the representation of functions very carefully because the greater part of the cooperation between Scheme and C depends on it.

Primitive functions with fixed arity are represented by objects referring to C functions of the same arity. Consequently, the function **cons** could be invoked from C in the form of the simple and mnemonic `SCM_cons(`x`,`y`)`.

First class functions in Scheme pose more serious problems. They can take any number of arguments. They can be computed. They may take their arguments in some special way if invoked by **apply**. Coming up with a way that is efficient and yet satisfies all these constraints is far from a trivial job. We'll allow ourselves a little more latitude here in showing several approaches, and for non-primitive functions, we'll adopt a way that is original and systematic (even if it's not very fast). [see **Ex. 10.1**]

Closures are represented by objects where the first field is a pointer to a C function. The second field indicates the arity of the function. The other supplementary fields contain closed variables. The macro `SCM_DefineClosure` defines an appropriate C structure for each type of closure.

```
#define SCM_DefineClosure(struct_name,fields) \
  struct struct_name {          \
    SCM (*behavior)(void);      \
    long arity;                 \
    fields }
```

When a closure is invoked by `SCM_invoke`, the closure receives itself as the first argument in order to extract its closed variables. When a function is called with an incorrect number of arguments, an error must be raised. We will assume that the function `SCM_invoke` takes care of this verification so that it doesn't encumber the one being called. However, when a function receives a variable number of arguments, we have to indicate to it how many to look for because there is no linguistic means of determining that in C. Thus we will assume that C functions associated with Scheme closures take the number of arguments to expect as their second argument. Those arguments come next, as **varargs** in traditional C or now renamed as **stdarg**. As we've already mentioned, this arrangement is not the best you could imagine; the one adopted for primitives is more efficient.

The names of C variables implemented afterwards (such as **self_**, **size_**, and **arguments_**) are for internal use and simply communicate with macros defining local variables. The only interesting case is that of a dotted variable which must package its arguments as a list. The function `SCM_list` offers the right interface for that purpose.

```
#define SCM_DeclareFunction(Cname) \
  SCM Cname (struct Cname *self_, unsigned long size_, \
```

```
            va_list arguments_)
#define SCM_DeclareLocalVariable(Cname,rank) \
  SCM Cname = va_arg(arguments_,SCM)
#define SCM_DeclareLocalDottedVariable(Cname,rank) \
  SCM Cname = SCM_list(size_-rank,arguments_)
#define SCM_Free(Cname) ((*self_).Cname)
```

One of the beauties of this approach is that access to closed variables is expressed very simply since they name the data members of the structure associated with the closure.

10.10 Execution Library

The execution library is a set of functions written in C. Those functions must be linked to a program so that the program can get resources that it still lacks. This library is skeletal in that it doesn't contain a lot of basic utility functions like `string-ref` or `close-output-port`. We're going to describe only the major representative functions and ignore the others (notably, `SCM_print` which appears in the generated `main`).

10.10.1 Allocation

There's no memory management, so there's no garbage collector either because building one would take too long. For more about that topic, you should consult [Spi90, Wil92]. We'll leave it as an exercise to adapt Boehm's garbage collector in [BW88] to what follows.

The most obvious allocation function is `cons`. With it, we allocate an object prefixed by its type. We fill in its type the same way that we fill in its `car` and `cdr`. Finally, we convert its address into an `SCM` for the return value.

```
SCM SCM_cons (SCM x, SCM y) {
  SCMref cell = (SCMref) malloc(sizeof(struct SCM_unwrapped_pair));
  if (cell == (SCMref) NULL) SCM_error(SCM_ERR_CANT_ALLOC);
  cell->pair.header.tag = SCM_PAIR_TAG;
  cell->pair.car = x;
  cell->pair.cdr = y;
  return SCM_Wrap(cell);
}
```

Closures are allocated by the function `SCM_close`. It takes a variable number of arguments, so *a posteriori* that justifies our choice about using multiple arguments in C. Thus we need to allocate an object of type `SCM_CLOSURE_TAG` and to fill in the other fields in terms of the number of arguments received.

```
SCM
SCM_close (SCM (*Cfunction)(void), long arity, unsigned long size, ...) {
  SCMref result = (SCMref) malloc(sizeof(struct SCM_unwrapped_closure)
                                  + (size-1)*sizeof(SCM) );
  unsigned long i;
  va_list args;
  if (result == (SCMref) NULL) SCM_error(SCM_ERR_CANT_ALLOC);
```

```
  result->closure.header.tag = SCM_CLOSURE_TAG;
  result->closure.behavior = Cfunction;
  result->closure.arity = arity;
  va_start(args,size);
  for ( i=0 ; i<size ; i++ ) {
    result->closure.environment[i] = va_arg(args,SCM);
  }
  va_end(args);
  return SCM_Wrap(result);
}
```

10.10.2 Functions on Pairs

The functions `car` and `set-car!` are simple, but they must not be confused with macros bearing similar names. These functions are safe in the sense that they test the types of their arguments before applying them. In general, there is a safe function and an unsafe function, and the latter should not be substituted for the former except when the compiler can be sure that the substitution is legitimate. Even though Lisp is not typed, its programs are such that almost two times out of three, type testing can be suppressed, according to [Hen92a, WC94, Ser94]. The attitude is that programmers are type-checking in their head, so a clever compiler can take advantage of that "preprocessing."

```
SCM SCM_car (SCM x) {
  if ( SCM_PairP(x) ) {
    return SCM_Car(x);
  } else return SCM_error(SCM_ERR_CAR);
}
SCM SCM_set_cdr (SCM x, SCM y) {
  if ( SCM_PairP(x) ) {
    SCM_Unwrap(x)->pair.cdr = y;
    return x;
  } else return SCM_error(SCM_ERR_SET_CDR);
}
```

It's also a good idea to understand the function `list`. It takes a variable number of arguments and thus receives them prefixed by that number. We have to admit that for its internal programming, we have once again sinned by resorting to recursion rather than iteration.

```
SCM SCM_list (unsigned long count, va_list arguments) {
  if ( count == 0 ) {
    return SCM_nil;
  } else {
    SCM arg = va_arg(arguments,SCM);
    return SCM_cons(arg,SCM_list(count-1,arguments));
  }
}
```

There is no error-trapping mechanism. We'll simply indicate errors by means of the C macro `SCM_error`. It conveys a representative error code (customary in C), the line number, and the file where the error occurred.

```
#define SCM_error(code) SCM_signal_error(code,__LINE__,__FILE__)
SCM SCM_signal_error (unsigned long code,
                      unsigned long line,
                      char *file ) {
  fflush(stdout);
  fprintf(stderr,"Error %u, Line %u, File %s.\n",code,line,file);
  exit(code);
}
```

10.10.3 Invocation

The two really important and subtle functions in the domain of invocations are `SCM_invoke` and `apply`. That's not surprising about `apply` since it has to know the calling protocol for functions in order to conform to it.

All the function calls other than those that are integrated (that is, transformed into a direct call to a macro or to the C function involved) pass through `SCM_invoke`. It takes the function to call as its first argument. As its second argument, it takes the number of arguments provided, and those arguments come next. As in the preceding interpreters, the function `SCM_invoke` is more or less generic. It analyzes its first argument to see that it is an invocable object: a primitive function or not (or an escape). It then extracts the object to apply and the arity of that object. Having extracted the arity, it compares the number of arguments it actually received. Finally, it passes these arguments to the function, following the appropriate protocol. Here we have three different protocols. There are others, such as suffixing the arguments by some constant such as `NULL` rather than prefixing them by their number. (That's what Bigloo does.)

- Primitives with fixed arity (like `SCM_cons`) are called directly with their arguments. The function call can be categorized as being of the type $f(x, y)$.
- Primitives with variable arity (like `SCM_list`) must necessarily know the number of arguments provided. That number will be passed as the first argument. The function call is thus of the type $f(n, x_1, x_2, \ldots, x_n)$.
- Closures must not only know the number of arguments received (when they have variable arity) but also they have to get their closed variables, stored, in fact, in the closure itself. The type of function call they produce will thus be of the type $f(f, n, x_1, x_2, \ldots, x_n)$.

Of course, we could refine, unify, or even eliminate some of these protocols, so here is the function `SCM_invoke`. In spite of its massive size, it is actually quite regular in structure. (We have withheld the part about continuations. We'll get to them in the next section.)

```
SCM SCM_invoke(SCM function, unsigned long number, ...) {
  if ( SCM_FixnumP(function) ) {
    return SCM_error(SCM_ERR_CANNOT_APPLY); /* Cannot apply a number! */
  } else {
    switch SCM_2tag(function) {
    case SCM_SUBR_TAG: {
      SCM (*behavior)(void) = (SCM_Unwrap(function)->subr).behavior;
```

```
long arity = (SCM_Unwrap(function)->subr).arity;
SCM result;
if ( arity >= 0 ) {          /* Fixed arity subr */
  if ( arity != number ) {
    return SCM_error(SCM_ERR_WRONG_ARITY); /* Wrong arity! */
  } else {
    if ( arity == 0) {
      result = behavior();
    } else {
      va_list args;
      va_start(args,number);
      { SCM a0 ;
        a0 = va_arg(args,SCM);
        if ( arity == 1 ) {
          result = ((SCM (*)(SCM)) *behavior)(a0);
        } else {
          SCM a1 ;
          a1 = va_arg(args,SCM);
          if ( arity == 2 ) {
            result = ((SCM (*)(SCM,SCM)) *behavior)(a0,a1);
          } else {
            SCM a2 ;
            a2 = va_arg(args,SCM);
            if ( arity == 3 ) {
              result = ((SCM (*)(SCM,SCM,SCM))
                        *behavior)(a0,a1,a2);
            } else {
              /* No fixed arity subr with more than 3 variables */
              return SCM_error(SCM_ERR_INTERNAL);
            }
          }
        }
      }
      va_end(args);
    }
    return result;
  }
} else {                     /* Nary subr */
  long min_arity = SCM_MinimalArity(arity) ;
  if ( number < min_arity ) {
    return SCM_error(SCM_ERR_MISSING_ARGS);
  } else {
    va_list args;
    SCM result;
    va_start(args,number);
    result = ((SCM (*)(unsigned long,va_list))
              *behavior)(number,args);
    va_end(args);
    return result;
  }
}
```

```
    }
    case SCM_CLOSURE_TAG: {
      SCM (*behavior)(void) = (SCM_Unwrap(function)->closure).behavior ;
      long arity = (SCM_Unwrap(function)->closure).arity ;
      SCM result;
      va_list args;
      va_start(args,number);
      if ( arity >= 0 ) {
        if ( arity != number ) { /* Wrong arity! */
          return SCM_error(SCM_ERR_WRONG_ARITY);
        } else {
          result = ((SCM (*)(SCM,unsigned long,va_list)) *behavior)
                   (function,number,args);
        }
      } else {
        long min_arity = SCM_MinimalArity(arity) ;
        if ( number < min_arity ) {
          return SCM_error(SCM_ERR_MISSING_ARGS);
        } else {
          result = ((SCM (*)(SCM,unsigned long,va_list)) *behavior)
                   (function,number,args);
        }
      }
      va_end(args);
      return result;
    }
    default: {
      SCM_error(SCM_ERR_CANNOT_APPLY);      /* Cannot apply! */
    }
    }
  }
}
```

The function **apply** is equally imposing in size. What's interesting about it is that since it is a primitive function of fixed arity, it gets its arguments as an instance of `va_list` where the last position contains a list that it must explore. The problem of interfacing multiple variables in C is that we can't construct new instances of `va_list` because it is a type that is private to the implementation of the C compiler. The only thing we can do is patiently accept this glitch and distinguish all the arities to generate all plausible calls. The boring part is that we can't enumerate all of them, so we must constrain **apply** to transmit only a limited number of arguments. That case was foreseen for COMMON LISP since there we have a constant—`call-arguments-limit`—to indicate the maximum number of arguments allowed. Its value should be at least 50. We'll restrict[10] ourselves to 14, of which only the first four possibilities are shown here.

```
SCM SCM_apply (unsigned long number, va_list arguments) {
  SCM args[31];
  SCM last_arg;
  SCM fun = va_arg(arguments,SCM);
```

10. If this limitation seems absurd to you, then test its value in your favorite Lisp.

```
    unsigned long i;
    for ( i=0 ; i<number-1 ; i++  ) {
      args[i] = va_arg(arguments,SCM);
    }
    last_arg = args[--i];
    while ( SCM_PairP(last_arg) ) {
      args[i++] = SCM_Car(last_arg);
      last_arg = SCM_Cdr(last_arg);
    }
    if ( ! SCM_NullP(last_arg) ) {
      SCM_error(SCM_ERR_APPLY_ARG);
    }
    switch ( i ) {
    case 0: return SCM_invoke(fun,0);
    case 1: return SCM_invoke(fun,1,args[0]);
    case 2: return SCM_invoke(fun,2,args[0],args[1]);
    case 3: return SCM_invoke(fun,3,args[0],args[1],args[2]);
    case 4: return SCM_invoke(fun,4,args[0],args[1],args[2],args[3]);
                    ...
    default: return SCM_error(SCM_ERR_APPLY_SIZE);
    }
  }
```

Since C doesn't support more than 32 arguments anyway, there is no need for as many as 50. Purists will notice the (unnecessary?) test verifying whether the last `cdr` of the list containing the final arguments submitted to `apply` is really empty.

The essential problem of invocation in C is that it doesn't preserve the property imposed by Scheme that a tail call should make a constant continuation. In consequence, a program translated into C can be interrupted because of stack overflow—a situation that could not occur with any standard implementation of Scheme. Some compilers from Scheme to C (such as Scheme→C or Bigloo) use a great deal of energy to avoid such problems. They look for recursive functions, loops, and so forth, but they cannot find every case and are thus vulnerable to certain programming styles. Another solution is not to use the C stack and instead to handle Scheme continuations explicitly.

10.11 `call/cc`: To Have and Have Not

We've just finished the compilation of a significant part of Scheme into C. In spite of the fact that we still lack many functions, the essentials are present (even if not very efficient). Continuations, in contrast, are still missing. As we did for the byte-code compiler, we'll first define the function `call/ep` to provide continuations with a dynamic extent, as in Lisp, and we'll translate that by `setjmp/longjmp`. Then in the second part of this section, we'll tackle the remaining issues about providing real continuations like in Scheme.

10.11.1 The Function call/ep

We described the function **call/ep** already in the same interface as the function **call/cc**, but the first class continuation that it synthesizes cannot legitimately be used except during its dynamic extent. [see p. 102] Here, we'll use the same implementation as in Chapter 7 to know that we've allocated an object on the heap to represent the continuation. That object will point to the **jmp_buf** in the stack, and that jump itself will refer to the continuation. Here are the functions involved:

```
SCM SCM_allocate_continuation (struct SCM_jmp_buf *address) {
  SCMref continuation =
          (SCMref) malloc(sizeof(struct SCM_unwrapped_escape));
  if (continuation == (SCMref) NULL) SCM_error(SCM_ERR_CANT_ALLOC);
  continuation->escape.header.tag = SCM_ESCAPE_TAG;
  continuation->escape.stack_address = address;
  return SCM_Wrap(continuation);
}
struct SCM_jmp_buf {
  SCM back_pointer;
  jmp_buf jb;
};
SCM SCM_callep (SCM f) {
  struct SCM_jmp_buf scmjb;
  SCM continuation = SCM_allocate_continuation(&scmjb);
  scmjb.back_pointer = continuation;
  if ( setjmp(scmjb.jb) != 0 ) {
    return jumpvalue;
  } else {
    return SCM_invoke1(f,continuation);
  }
}
```

When a continuation is invoked, the function **SCM_invoke** perceives it and verifies the arity of this call before going on to the call itself. The following fragment should be inserted in the function **SCM_invoke** as we've already defined it. To verify that the continuation is valid (and because we have no **unwind-protect** to invalidate it), we test whether it matches the **jmp_buf** in the stack and whether we really are *above* that **jmp_buf**. Unfortunately, that second verification requires us to know the direction of the C stack. That information is packaged in the macro **SCM_STACK_HIGHER**. The definition of that macro depends on the implementation, of course, but we can get information about it from a simple little program in portable C. Generally, with UN*X, the stack grows with decreasing addresses, so **SCM_STACK_HIGHER** is none other than <=.

```
...  case SCM_ESCAPE_TAG: {
       if ( number == 1) {
         va_list args;
         va_start(args,number);
         jumpvalue = va_arg(args,SCM);
         va_end(args);
         { struct SCM_jmp_buf *address =
             SCM_Unwrap(function)->escape.stack_address;
```

```
            if ( SCM_EqP(address->back_pointer,function)
                && ( (void *) &address
                        SCM_STACK_HIGHER (void *) address ) ) {
              longjmp(address->jb,1);
            } else {    /* surely out of dynamic extent! */
              return SCM_error(SCM_ERR_OUT_OF_EXTENT);
            }
          }
        } else {
          /* Not enough arguments! */
          return SCM_error(SCM_ERR_MISSING_ARGS);
        }
      } ...
```

What we said about the efficiency of `call/ep` in the context of the byte-code compiler no longer holds true here because in C `longjmp` is notoriously slow and thus enormously delays programs that use it too much. The point for us, however, is to show how well we can integrate Lisp and C, so we'll live with it.

10.11.2 The Function `call/cc`

Alas! if you're nostalgic for real continuations, then you are probably still longing for them at this point. We know that we can get `call/cc` as a magic function, mysterious and obscure, but unfortunately in order to *write* it, we need to know at least a little about the C stack because that stack contains what we want to capture. Unfortunately, C doesn't really offer a *portable* means of inspecting the stack so it is extremely hard to implement this type of continuation in portable C without loading ourselves down with conditional definitions. Our only choice, then, is to use CPS to transform a program so that continuations appear and then compile it all with the preceding compiler.

Making Continuations Explicit

The transformation we'll offer is equivalent to the one we presented earlier and once again uses our favorite code walker. [see p. 177] This transformation works on the initial expression, once it has been objectified. However, since the `let` forms (that is, nodes of the class `Fix-Let`) will have disappeared in this affair, we will re-introduce them by means of a second code walk to retrieve and convert closed forms. In passing, it will suppress the infamous administrative redexes. (Just after CPS, we'll explain the transformation `letify`.) Here's the new version of the compiler:

```
(define (compile->C e out)
  (set! g.current '())
  (let* ((ee (letify (cpsify (Sexp->object e)) '()))
         (prg (extract-things! (lift! ee))) )
    (gather-temporaries! (closurize-main! prg))
    (generate-C-program out e prg) ) )
```

The code walker that makes continuations explicit is called `cpsify`. It results from the interaction between the function `update-walk!` and the generic function

->CPS. Two new classes of objects are introduced: the class of continuations (serving only to mark abstractions that play the role of continuations) and the class of pseudo-variables (serving as variables for continuations).

```
(define-class Continuation Function ())
(define-class Pseudo-Variable Local-Variable ())
(define (cpsify e)
  (let ((v (new-Variable)))
    (->CPS e (make-Continuation (list v) (make-Local-Reference v))) ) )
(define new-Variable
  (let ((counter 0))
    (lambda ()
      (set! counter (+ 1 counter))
      (make-Pseudo-Variable counter #f #f) ) ) )
```

The function ->CPS takes the object to convert as its first argument and the current continuation as its second. By default, it applies the continuation to the object. The initial continuation appearing in **cpsify** is the objectified identity, $\lambda x.x$.

```
(define-generic (->CPS (e Program) k)
  (convert2Regular-Application k e) )
(define (convert2Regular-Application k . args)
  (make-Regular-Application k (convert2arguments args)) )
```

Now all we have to do is to articulate the appropriate methods. For sequences, that's simple: we convert the first form and relegate the second one to the continuation, like this:

```
(define-method (->CPS (e Sequence) k)
  (->CPS (Sequence-first e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (->CPS (Sequence-last e) k) ) ) ) )
```

For alternatives, the method is also simple, but the continuation should be duplicated in both branches of an alternative, like this:

```
(define-method (->CPS (e Alternative) k)
  (->CPS (Alternative-condition e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (make-Alternative
                      (make-Local-Reference v)
                      (->CPS (Alternative-consequent e) k)
                      (->CPS (Alternative-alternant e) k) ) ) ) ) )
```

Assignments are also straightforward. We convert the value to assign while being careful to make the modification in the continuation, like this:

```
(define-method (->CPS (e Box-Write) k)
  (->CPS (Box-Write-form e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (convert2Regular-Application
                      k (make-Box-Write
```

```
                        (Box-Write-reference e)
                        (make-Local-Reference v) ) ) ) ) ) )
(define-method (->CPS (e Global-Assignment) k)
  (->CPS (Global-Assignment-form e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (convert2Regular-Application
                      k (make-Global-Assignment
                         (Global-Assignment-variable e)
                         (make-Local-Reference v) ) ) ) ) ) )
```

To simplify our lives (and because objectification is overdoing things), we undo closed applications in favor of an application of a closure. You recall that closed applications generated by transformations are identified by `letify`. There will be a great many of those by the way because the naïve CPS transformation that we programmed generates many *administrative* redexes [SF92]. Those redexes are not too troublesome, however, because if they are compiled correctly, they will just naturally go away; they may be ugly to look at, but they interfere only with interpretation.

```
(define-method (->CPS (e Fix-Let) k)
  (->CPS (make-Regular-Application
          (make-Function (Fix-Let-variables e) (Fix-Let-body e))
          (Fix-Let-arguments e) )
         k ) )
```

Functions will be burdened with another argument, one to represent the continuation of their caller.

```
(define-method (->CPS (e Function) k)
  (convert2Regular-Application
   k (let ((k (new-Variable)))
       (make-Function (cons k (Function-variables e))
                      (->CPS (Function-body e)
                             (make-Local-Reference k) ) ) ) ) )
```

Functional applications are more complicated forms that have to be handled in a similar way. We must compute all the arguments, one after another, before applying the function to them. Once again, in doing that, we've chosen left to right order.

```
(define-method (->CPS (e Predefined-Application) k)
  (let* ((args (Predefined-Application-arguments e))
         (vars (let name ((args args))
                 (if (Arguments? args)
                     (cons (new-Variable)
                           (name (Arguments-others args)) )
                     '() ) ))
         (application
          (convert2Regular-Application
           k
           (make-Predefined-Application
            (Predefined-Application-variable e)
            (convert2arguments
```

```
                (map make-Local-Reference vars) ) ) ) ) )
      (arguments->CPS args vars application) ) )
  (define-method (->CPS (e Regular-Application) k)
    (let* ((fun (Regular-Application-function e))
           (args (Regular-Application-arguments e))
           (varfun (new-Variable))
           (vars (let name ((args args))
                   (if (Arguments? args)
                       (cons (new-Variable)
                             (name (Arguments-others args)) )
                       '() ) ))
           (application
            (make-Regular-Application
             (make-Local-Reference varfun)
             (make-Arguments k (convert2arguments
                                (map make-Local-Reference vars) ) ) ) ) )
      (->CPS fun (make-Continuation
                  (list varfun)
                  (arguments->CPS args vars application) )) ) )
  (define (arguments->CPS args vars appl)
    (if (pair? vars)
        (->CPS (Arguments-first args)
               (make-Continuation
                (list (car vars))
                (arguments->CPS (Arguments-others args)
                                (cdr vars)
                                appl ) ) )
        appl ) )
```

Re-introducing Closed Forms

The `letify` transformation indicated earlier has the responsibility of identifying closed forms and translating them into appropriate `let` forms. However, we're going to take advantage of a little polishing here to clean up the result of `->CPS`. When `->CPS` handles an alternative, it will duplicate the same subtree of abstract syntax representing the continuation—by sharing it physically—in both branches of the alternative. Thus we no longer have a tree of abstract syntax, but rather a DAG (directed acyclic graph). To avoid having the eventual transformations fumble because of these hidden physical sharings, the function `letify` will entirely copy the DAG of abstract syntax into a pure tree of abstract syntax. We assume that we have the generic function `clone` available for that. **[see Ex. 11.2]** It should be able to duplicate any MEROONET object, and we'll adapt it to the case of variables that must be renamed. So here is that transformation. With `collect-temporaries!`, it has a certain resemblance to renaming local variables.

```
(define-generic (letify (o Program) env)
  (update-walk! letify (clone o) env) )
(define-method (letify (o Function) env)
  (let* ((vars (Function-variables o))
         (body (Function-body o))
```

```
          (new-vars (map clone vars)) )
      (make-Function
       new-vars
       (letify body (append (map cons vars new-vars) env)) ) ) )
(define-method (letify (o Local-Reference) env)
  (let* ((v (Local-Reference-variable o))
         (r (assq v env)) )
    (if (pair? r)
        (make-Local-Reference (cdr r))
        (letify-error "Disappeared variable" o) ) ) )
(define-method (letify (o Regular-Application) env)
  (if (Function? (Regular-Application-function o))
      (letify (process-closed-application
               (Regular-Application-function o)
               (Regular-Application-arguments o) )
              env )
      (make-Regular-Application
       (letify (Regular-Application-function o) env)
       (letify (Regular-Application-arguments o) env) )  ) )
(define-method (letify (o Fix-Let) env)
  (let* ((vars (Fix-Let-variables o))
         (new-vars (map clone vars)) )
    (make-Fix-Let
     new-vars
     (letify (Fix-Let-arguments o) env)
     (letify (Fix-Let-body o)
             (append (map cons vars new-vars) env) ) ) ) )
(define-method (letify (o Box-Creation) env)
  (let* ((v (Box-Creation-variable o))
         (r (assq v env)) )
    (if (pair? r)
        (make-Box-Creation (cdr r))
        (letify-error "Disappeared variable" o) ) ) )
(define-method (clone (o Pseudo-Variable))
  (new-Variable) )
```

Execution Library

It's probably hard for you to see where the preceding transformation is going. Its goal is to make continuations apparent, that is, to make `call/cc` trivial to implement. A continuation will be represented by a closure that forgets the current continuation in order to restore the one that it represents. Here's the definition of `SCM_callcc`. Since it is a normal function, it is called with the continuation of the caller as its first argument, so here its arity is two! The function `SCM_invoke_continuation` appears first to help the C compiler, which likes to find things in the right order.

```
SCM SCM_invoke_continuation (SCM self, unsigned long number,
                             va_list arguments) {
  SCM current_k = va_arg(arguments,SCM);
```

```
  SCM value     = va_arg(arguments,SCM);
  return SCM_invoke1(SCM_Unwrap(self)->closure.environment[0],value);
}

SCM SCM_callcc (SCM k, SCM f) {
  SCM reified_k =
    SCM_close(SCM_CfunctionAddress(SCM_invoke_continuation),
              2, 1, k);
  return SCM_invoke2(f,k,reified_k);
}

SCM_DefinePredefinedFunctionVariable(CALLCC,"CALL/CC",2,SCM_callcc);
```

The library of predefined functions in C has not changed. In contrast, their call protocol has been modified somewhat because of these swarms of continuations. If we write `(let ((f car)) (f '(a b)))`, then our dumb compiler does not realize that it is in fact equivalent to `(car '(a b))` (or even directly equivalent to `(quote a)` by propagating constants). It will generate a computed call to the value of the global variable `CAR`, which will receive a continuation as its first argument and a pair as its second. The new value of `CAR` is, in fact, merely `(lambda (k p) (k (car p)))` expressed with the old function `car`. Consequently, we will introduce a few C macros again here, and we will redefine the initial library with an arity increased by one. So that we don't leave you behind here, we'll show you a few examples. The functions prefixed by `SCMq_` make up the interface to functions prefixed by `SCM_`.

```
#define SCM_DefineCPSsubr2(newname,oldname)      \
  SCM newname (SCM k, SCM x, SCM y) {            \
    return SCM_invoke1(k,oldname(x,y));          \
  }
#define SCM_DefineCPSsubrN(newname,oldname)                        \
  SCM newname (unsigned long number, va_list arguments) {          \
    SCM k = va_arg(arguments,SCM);                                 \
    return SCM_invoke1(k,oldname(number-1,arguments));             \
}
SCM_DefineCPSsubr2(SCMq_gtp,SCM_gtp)
SCM_DefinePredefinedFunctionVariable(GREATERP,">",3,SCMq_gtp);
SCM_DefineCPSsubrN(SCMq_list,SCM_list)
SCM_DefinePredefinedFunctionVariable(LIST,"LIST",-2,SCMq_list);
```

One last problem is what to do with `apply`. We have to re-arrange its arguments so that the continuation gets them in the right positions.

```
SCM SCMq_apply (unsigned long number, va_list arguments) {
  SCM args[32];
  SCM last_arg;
  SCM k   = va_arg(arguments,SCM);
  SCM fun = va_arg(arguments,SCM);
  unsigned long i;
  for ( i=0 ; i<number-2 ; i++  ) {
    args[i] = va_arg(arguments,SCM);
  }
  last_arg = args[--i];
```

```
  while ( SCM_PairP(last_arg) ) {
    args[i++] = SCM_Car(last_arg);
    last_arg = SCM_Cdr(last_arg);
  }
  if ( ! SCM_NullP(last_arg) ) {
    SCM_error(SCM_ERR_APPLY_ARG);
  }
  switch ( i ) {
  case 0: return SCM_invoke(fun,1,k);
  case 1: return SCM_invoke(fun,2,k,args[0]);
  case 2: return SCM_invoke(fun,3,k,args[0],args[1]);
  ...
   default: return SCM_error(SCM_ERR_APPLY_SIZE);
  }
}
```

Example

Let's take our current example and look at the generated code. In spite of the C syntax, you can still see the "color" of CPS in it.

o/chap10kex.c

```
 2 /* Compiler to C $Revision: 4.0 $
 3 (BEGIN
 4   (SET! INDEX 1)
 5   ((LAMBDA
 6      (CNTER . TMP)
 7      (SET! TMP (CNTER (LAMBDA (I) (LAMBDA X (CONS I X)))))
 8      (IF CNTER (CNTER TMP) INDEX))
 9     (LAMBDA (F) (SET! INDEX (+ 1 INDEX)) (F INDEX))
10     'FOO))  */
11
12 #include "scheme.h"
13
14 /* Global environment: */
15 SCM_DefineGlobalVariable(INDEX,"INDEX");
16
17 /* Quotations: */
18 #define thing3 SCM_nil /* () */
19 SCM_DefineString(thing4_object,"FOO");
20 #define thing4 SCM_Wrap(&thing4_object)
21 SCM_DefineSymbol(thing2_object,thing4);       /* FOO */
22 #define thing2 SCM_Wrap(&thing2_object)
23 #define thing1 SCM_Int2fixnum(1)
24 #define thing0 thing1 /* 1 */
25
26 /* Functions: */
27 SCM_DefineClosure(function_0, SCM I; );
28
```

```
29 SCM_DeclareFunction(function_0) {
30   SCM_DeclareLocalVariable(v_25,0);
31   SCM_DeclareLocalDottedVariable(X,1);
32   SCM v_27; SCM v_26;
33   return (v_26=SCM_Free(I),
34           (v_27=X,
35            SCM_invoke1(v_25,
36                        SCM_cons(v_26,
37                                 v_27))));
38 }
39
40 SCM_DefineClosure(function_1, );
41
42 SCM_DeclareFunction(function_1) {
43   SCM_DeclareLocalVariable(v_24,0);
44   SCM_DeclareLocalVariable(I,1);
45   return SCM_invoke1(v_24,
46                      SCM_close(SCM_CfunctionAddress
47                                (function_0),-2,1,I));
48 }
49
50 SCM_DefineClosure(function_2, SCM v_15; SCM CNTER; SCM TMP; );
51
52 SCM_DeclareFunction(function_2) {
53   SCM_DeclareLocalVariable(v_21,0);
54   SCM v_20; SCM v_19; SCM v_18; SCM v_17;
55   return (v_17=(SCM_Content(SCM_Free(TMP))=v_21),
56           (v_18=SCM_Free(CNTER),
57            ((v_18 != SCM_false)
58             ? (v_19=SCM_Free(CNTER),
59                (v_20=SCM_Content(SCM_Free(TMP)),
60                 SCM_invoke2(v_19,
61                             SCM_Free(v_15),
62                             v_20)))
63             : SCM_invoke1(SCM_Free(v_15),
64                           SCM_CheckedGlobal(INDEX)))));
65 }
66
67 SCM_DefineClosure(function_3, );
68
69 SCM_DeclareFunction(function_3) {
70   SCM_DeclareLocalVariable(v_15,0);
71   SCM_DeclareLocalVariable(CNTER,1);
72   SCM_DeclareLocalVariable(TMP,2);
73   SCM v_23; SCM v_22; SCM v_16;
74   return (v_16=TMP= SCM_allocate_box(TMP),
75           (v_22=CNTER,
76            (v_23=SCM_close(SCM_CfunctionAddress(function_1),2,0),
77             SCM_invoke2(v_22,
78                         SCM_close(SCM_CfunctionAddress
79                                   (function_2),1,3,v_15,CNTER,TMP),
```

```
80                          v_23))));
81 }
82
83 SCM_DefineClosure(function_4, );
84
85 SCM_DeclareFunction(function_4) {
86   SCM_DeclareLocalVariable(v_8,0);
87   SCM_DeclareLocalVariable(F,1);
88   SCM v_11; SCM v_10; SCM v_9; SCM v_12; SCM v_14; SCM v_13;
89   return (v_13=thing1,
90           (v_14=SCM_CheckedGlobal(INDEX),
91            (v_12=SCM_Plus(v_13,
92                           v_14),
93             (v_9=(INDEX=v_12),
94              (v_10=F,
95               (v_11=SCM_CheckedGlobal(INDEX),
96                SCM_invoke2(v_10,
97                            v_8,
98                            v_11)))))));
99 }
100
101 SCM_DefineClosure(function_5, );
102
103 SCM_DeclareFunction(function_5) {
104   SCM_DeclareLocalVariable(v_1,0);
105   return v_1;
106 }
107
108 SCM_DefineClosure(function_6, );
109
110 SCM_DeclareFunction(function_6) {
111   SCM v_5; SCM v_7; SCM v_6; SCM v_4; SCM v_3; SCM v_2; SCM v_28;
112   return (v_28=thing0,
113           (v_2=(INDEX=v_28),
114            (v_3=SCM_close(SCM_CfunctionAddress(function_3),3,0),
115             (v_4=SCM_close(SCM_CfunctionAddress(function_4),2,0),
116              (v_6=thing2,
117               (v_7=thing3,
118                (v_5=SCM_cons(v_6,
119                              v_7),
120                 SCM_invoke3(v_3,
121                             SCM_close(SCM_CfunctionAddress
122                                       (function_5),1,0),
123                             v_4,
124                             v_5)))))))));
125 }
126
127
128 /* Expression: */
129 void main(void) {
130   SCM_print(SCM_invoke0(SCM_close(SCM_CfunctionAddress
```

```
131                                    (function_6),0,0)));
132   exit(0);
133 }
134
135 /* End of generated code. */
136
```

You see that the size of the file we produce grows from 76 to 130 lines. The number of functions generated by compilation also increases from 3 to 6, and the number of local variables used increases from 2 to 22. In short, the executable has fattened up a bit. The computation takes a little longer, too. (It took about 50% longer on our machine.) Afterwards, we modified `main` to repeat the computation 10 000 times (without calling `SCM_print`), and we compiled with `-O` as the optimization level. The computation then increased from 1.1 seconds to 1.7. In both cases, the computation consumes the same amount of space for continuations, but in the first case, the allocations occur on the stack (where the hardware and the C compiler provide ingenious treasures for us) whereas in the second case, allocations occur on the heap and destroy the locality of its references. As shown in [AS94], we could, of course, overcome most of these disadvantages.

```
% gcc -ansi -pedantic chap10kex.c scheme.o schemeklib.o
% time a.out
(2 3)
0.000u 0.020s 0:00.00 0.0% 3+3k 0+0io 0pf+0w
% size a.out
text    data    bss     dec     hex
32768   4096    32      36896   9020
```

The transformation that we just described does not get around the problem of stack overflow due to our not preserving the property of tail recursion. In fact, it is actually made worse by the transformation since the transformation produces more applications than before but never comes back to these applications. Functions are called but never return a result, so the C stack will overflow eventually if the computation is not completed before. One solution is to make a `longjmp` from time to time, just to lower the stack, as in [Bak].

10.12 Interface with C

Since our little compiler can represent data, and in addition, it has adopted the calling protocol for C functions, it is particularly well suited to use with C. This *foreign interface* is very important because it can take advantage of huge libraries of utilities written in C. We'll show you an example of such a foreign interface, one chosen to illustrate both its usefulness and the associated problems.

The UN*X `system` function takes a character string, hands it to the command interpreter (`sh`), and then returns an integer corresponding to its return code. We'll assume that we have a macro available—`defforeignprimitive`—to declare the interface for this `system` function to the compiler, like this:

```
(defforeignprimitive system int ("system" string) 1)
```

As planned, when the compiler sees (`system` π), the compiler will verify that its argument is a character string; then it will call the `system` function, and eventually it will convert the result (a C integer) into a Scheme integer. Since C strings and Scheme strings are coded the same way, it's easy to convert one to the other, but other conversions can pose strenuous problems, according to [RM92, DPS94a].

Of course, that conversion is not quite enough. We must again insure that the `system` function can be called by a computation or by `apply`. That condition implies that there must be a Scheme value representing the function, but we'll ignore that delicate problem.

10.13 Conclusions

This chapter presented a compiler from Scheme to C. We could adapt that compiler to the execution library of evaluators written in C, such as Bigloo [Ser94], SIOD [Car94], or SCM [Jaf94]. In this chapter, you've been able to see the problems of compilation into a high-level language, the gap that separates Scheme from C, and also the benefits of reasonable cooperation between the two languages.

We strongly urge you to compare the compiler towards C with the compiler into byte-code. We could readily marry the techniques you saw earlier with the new ones here, for example:

- changing the compilation of quotations so that they would be read by `read`;
- freeing ourselves from the C stack (and from C calling protocol) to take advantage of a stack dedicated to Scheme;
- extending the compiler to compile independent modules; and so on. [see p. 223]

We could also measure the cost (in the size of the execution library and in the speed of computations) of adding a function such as `load`, or more simply, `eval`. We'll leave how to incorporate them as a (strenuous) exercise. And we'll also leave as an exercise just how to bootstrap the evaluator, once it's been stuffed like that.

10.14 Exercises

Exercise 10.1 : Invoking closures could borrow the same technique as the one used for predefined functions of fixed arity; that is, it could adopt a model that could be characterized as $f(f, x, y)$. Modify whatever needs to be changed to improve the compiler in that way.

Exercise 10.2 : Access to global variables would be more efficient if we suppressed the test `SCM_CheckedGlobal` on reading. That test verifies whether the variable has been initialized. Design another analysis of initialization. In doing so, look for a way to characterize the references for which you can be sure that the variable has been initialized.

Project 10.3 : The code produced by the compiler in this chapter generates C that conforms to ISO 90 [ISO90]. Modify whatever is necessary to generate Kernighan & Ritchie C as in [KR78].

Project 10.4 : Adapt the code generator in this chapter to produce C++, as in [Str86], rather than C.

Recommended Reading

In recent years, there has been a lot of good work about compiling into C. Nitsan Séniak treats the topic excellently in his thesis [Sén91], as does Manuel Serrano in [Ser94]. If you are completely enamored of this subject, you can happily lose yourself in the source code for Bigloo [Ser94].

11
Essence of an Object System

OBJECTS! Oh, where would we be without them? This chapter defines the object system that we've used throughout this book. We deliberately restricted ourselves to a limited set of characteristics so that we would not overburden the description which will follow here. In fact, as Rabelais would say, we want to limit it to its *sustantificque mouelle*, that is, to its very essence.

This object system is called MEROON.[1] Such a system is complicated and demands considerable attention if we want it to be simultaneously efficient and portable. As a result, the system is endowed with a structure strongly influenced, maybe even distorted, by our worries about portability. To compensate for that, we're actually going to show you a reduced version of MEROON, and we'll call that reduced version[2] MEROONET.

Lisp and objects have a long history in common. It begins with one of the first object languages, Smalltalk 72, which was first implemented in Lisp. Since that time, Lisp, as an excellent development language, has served as the cradle for innumerable studies about objects. We'll mention only a couple: Flavors, developed by Symbolics for a windowing system on Lisp machines, experimented with multiple inheritance; Loops, created at Xerox Parc, introduced the idea of generic functions. Those efforts culminated in the definition of CLOS (COMMON LISP Object System) and of ΤΕΛΟΣ (the EULISP object system). These latter two systems bring together most of the characteristics of the preceding object systems while immersing them in the typing system of the underlying language. By doing so, they satisfy the first rule of one of these systems: everything should be an object.

Compared to object systems you find in other languages, the object systems of Lisp are distinguished in two ways:

- Generic functions and their multimethods, a technique known as *multiple dispatch*.

  Sending a message, written as (*message arguments* ...), is not distinguished syntactically from calling a function, written as (*function arguments*...).

1. This name came from my son's teddy bear, but you may try to give a sensible meaning to that acronym.

2. Both these systems—MEROON and MEROONET—are available by anonymous `ftp` according to instructions on page xix.

Multimethods do not assume that the receiver of a message is unique, even though that is often the case. Rather, they determine the method as a function of the class of the significant arguments. For example, printing a number in hexadecimal on a stream is not a method for numbers, nor a method for streams, but a method working on the Cartesian product of number × stream.

- Reflection.

  Reflection is the quality of systems endowed with the power to speak of themselves. These systems introduce the idea of *metaclasses*. The class of an object is itself an object and thus necessarily belongs to a class which we call a *metaclass*, which itself is also an object, and thus it goes on. The behavior of these meta-objects is known as the *meta-object protocol* or MOP. It controls the possibilities of the object system. Levels of reflection are possible, for example, from self-description to self-modification. It thus becomes possible to specify the physical representation of objects while still conserving their inherited properties so, for example, we can accomodate a persistent archiving system or a data base. Self-description is an important issue for introspection, notably for design tools to implement programs since such tools make it possible to inspect objects and to know the structure of objects so we can print them, compile them, [see p. 340], make them migrate from one machine to another, as in [Que94], and so forth.

We're extending that long history of object systems as we present MEROONET. We hope that any restrictions we imposed in building MEROONET will not overshadow the excellent qualities that it still has. Among them are the following:

- All values that can be handled in Lisp or Scheme—including vectors—can be represented by MEROONET objects without restrictions on their inheritance.
- MEROONET is self-describing because classes are objects—whole, apart, and duly available for inspection. We've avoided the trap of infinite regression as they did in ObjVlisp, according to [Coi87, BC87].
- There are generic functions, like in CLOS [BDG+88], but without multimethods.
- The code is highly efficient.

A number of implementations of object systems in Lisp or Scheme have already been described in the literature, among them [AR88, Kes88, Coi87, MNC+89, Del89, KdRB92]. However, as a system, MEROONET differs from those in several ways:

- MEROONET, as we hinted, has adopted the generic functions of Common Loops [BKK+86] but not multimethods.
- MEROONET supports classes as first class objects, as in ObjVlisp. This makes a great deal of self-description possible.
- MEROONET abhors multiple inheritance! When we know more about the semantics of multiple inheritance and when the problems posed in [DHHM92] have been resolved, then MEROONET will change its mind about this issue—maybe.

- Objects are represented as contiguous values, that is, as vectors. This representation makes it possible to define data imported from other languages and to do so by means of new field descriptors, as in [QC88, Que90a, Que95].

As we describe MEROONET, we'll also discuss the reasons for our implementation choices. Some of those choices were due to the implementation language, Scheme. Those choices, of course, might have been different if we had directly implemented MEROONET in C, for example. In trying to simplify our explanation of MEROONET, we will cover it from bottom to top, gradually introducing functions as they are needed. Documentation about how to use MEROONET is mixed in with the implementation, but you can also consult the short version of the user's manual (which we won't bother to repeat here). **[see p. 87]**

11.1 Foundations

The first implementation decision involves object representation in MEROONET. For that, we use sets of contiguous values. To express that in Scheme, we chose vectors. Within a vector, we reserve the first index (that is, index zero) to store the class identity, so that we can access its class from an object. That kind of relation is known as an *instantiation link*. It would be more obvious to have an object point directly to its class (that is, to the most specific class to which it belongs), but we prefer to number the classes and have each object store one such number. Printing objects by the usual means in Scheme thus becomes easier because classes are generally large[3] and unweildy data structures whose details don't mean much to most of us. They may also include circular references that make the normal way of printing with `display` cryptic or illegible. Later, when we look at the means for calling generic functions and the test for belonging to a class, you'll see other reasons for using these numbers as indices to classes.

If we number the classes, we have to be able to convert such a number into a class. All classes will consequently be archived in a vector. Since vectors cannot be extended in Scheme (in contrast to COMMON LISP) there will be a maximum number of possible classes but not explicit detection of the anomaly if the number of classes exceeds this limit. The variable `*class-number*` indicates the first free number in the vector `*classes*`. Since some classes are predefined from the moment of bootstrapping, this number will increase. Since we don't want to burden the code with overly scrupulous tests, we won't bother to test whether a number really indicates a class.

```
(define *maximal-number-of-classes* 100)
(define *classes* (make-vector *maximal-number-of-classes* #f))
(define (number->class n)
  (vector-ref *classes* n) )
(define *class-number* 0)
```

Handling classes by their numbers is somewhat laconic, so it obliges us to name all the classes. Consequently, there are no anonymous classes. Rather, classes are

3. For some garbage collectors, if all the values of Scheme are objects, one pointer less per object to inspect or to update might also be advantageous.

seen as static entities. That is, they are not created dynamically. The function `->Class` converts a symbol into a class of that name. Again, not wanting to overburden the code with superfluous tests, we make the function `->Class` return the most recent class by the name we're looking for. That convention will help us redefine classes without, however, giving redefinition too precise a meaning, so we should refrain from redefining the same class many times without a good reason.

```
(define (->Class name)
  (let scan ((index (- *class-number* 1)))
    (and (>= index 0)
         (let ((c (vector-ref *classes* index)))
           (if (eq? name (Class-name c))
               c (scan (- index 1)) ) ) ) ) )
```

For generic functions, the implementation demands that they should have names and that we can access their internal structure. For those reasons, we'll keep a list of generic functions: `*generics*`. The function `->Generic` will convert such a name into a generic function. (We'll come back to this point later.)

```
(define *generics* (list))
(define (->Generic name)
  (let lookup ((l *generics*))
    (if (pair? l)
        (if (eq? name (Generic-name (car l)))
            (car l)
            (lookup (cdr l)) )
        #f ) ) )
```

11.2 Representing Objects

As we've indicated, objects in MEROONET are sets of contiguous values. Since we want every Scheme value to be assimilated by an object defined in MEROONET, we need to take into account the idea of a vector,[4] and thus we introduce the idea of an indexed field which contains a certain number of values, rather than containing a unique value. That characteristic existed already in Smalltalk [GR83], but in such a way that it was not possible to inherit a class cleanly if it had an indexed field. MEROONET does not have this kind of limitation.

There is a problem with character strings: for reasons of efficiency, they are usually primitives in a programming language. Nevertheless, we could represent them by vectors of characters—not such a bad idea if we are concerned about internationalization where we might be tempted to adopt larger and larger character sets represented by two or even four bytes. However, if we still want characters to be only one byte, and since Scheme doesn't support data structures mixing repetitions of such characters (of one byte) with other values (such as pointers), then MEROONET cannot efficiently simulate character strings.

MEROONET offers two kinds of fields: normal fields (defined by objects of the class `Mono-Field`) and indexed fields (defined by the class `Poly-Field`). A normal

4. In fact, we may even want to simulate vectors by MEROONET objects, which are themselves represented by vectors. The cost is just one coordinate to contain the instantiation link.

field will be implemented as a component of a vector, whereas an indexed field will be represented rather like strings in Pascal, that is, by a set of indexed values. This convention means that the representation must be prefixed by the number of its components. To clarify these ideas, consider the class of points defined like this:

```
(define-class Point Object (x y))
```

Let's assume that the number 7 is associated with the class **Point**, so the value as a point of **(make-Point 11 22)** will be represented by the vector **#(7 11 22)**. A polygon could be defined as a set of points representing the various segments that form its sides. We could make **Polygon** inherit **Point** to fix its point of reference.

```
(define-class Polygon Point ((* side)))
```

Here you see another kind of possible syntax—parenthesized—to define fields. A normal field will be prefixed by an equal sign whereas an indexed field will be prefixed by an asterisk.[5] If we want colored polygons, we'll create a new subclass, like this:

```
(define-class ColoredPolygon Polygon ((= color)))
```

Every class definition gives rise to a host of functions, notably, a function for creating objects. Its name is formed by prefixing **make-** to the name of the class. When we create a class, we must define all its fields. In particular, we have to define the size of each indexed field, so we will prefix the indexed values appearing in such fields by their number. To define a triangle, that is, a three-sided polygon, and to color it orange, we'll write it like this and then study its representation as vectors:

```
(make-ColoredPolygon
 11                                                    ;x
 22                                                    ;y
 3 (make-Point 44 55) (make-Point 66 77) (make-Point 88 99)  ;3 sides
 'orange )                                             ;color
→ #(9              ;(Class-number ColoredPolygon-class)
      11           ;x
      22           ;y
      3            ;number of sides
      #(7 44 55)   ;side[0]
      #(7 66 77)   ;side[1]
      #(7 88 99)   ;side[2]
      orange )     ;color
```

The objects of MEROONET are thus all represented by vectors where the first component contains the number of the class of that object. To keep our terminology straight, we'll use the word *offset* when we're talking about the offset within a vector representing an object, and we'll say *index* when we're handling an indexed field. Since one component is reserved for the instantiation link, the first valid offset is given by the constant ***starting-offset***. You can find out the class of an object by means of the function **object->class**.[6] We recognize whether a value is an

5. We chose the asterisk to indicate multiplicity, like Kleene used in the notation of regular expressions.

6. That function is equivalent to the function **class-of** in COMMON LISP, EULISP, and IS-Lisp. We named it differently to avoid introducing confusion and thus be able, for example, to mix multiple object systems.

object of MEROONET by means of the function `Object?`. That function poses a serious problem since Scheme does not allow the creation of new data types. The predicate `Object?` recognizes all MEROONET objects, but unfortunately, it also recognizes vectors of integers, etc.

To complete this prelude, here are the naming conventions for the variables we've used:

`o`, `o1`, `o2` ...	object
`v` ...	value (integer, pair, closure, etc.)
`i` ...	index

```
(define *starting-offset* 1)
(define (object->class o)
  (vector-ref *classes* (vector-ref o 0)) )
(define (Object? o)
  (and (vector? o)
       (integer? (vector-ref o 0)) ) )
```

11.3 Defining Classes

To define a class, there is the form `define-class`, taking three successive arguments:

- the name of the class to define;
- the name of its superclass;
- the specification of its own fields.

The class that will be created will inherit fields from the definition of its superclass. That kind of inheritance is known as *field inheritance*. A class also inherits all the behavior of its superclass with respect to existing generic functions. This kind of inheritance is known as *method inheritance*.

Here's the syntax of the form `define-class`:

`(define-class` *class-name superclass-name* `(` *list-of-fields* `) )`

A field appearing in the list of fields can be mentioned either directly by name (in that case, it's a normal field) or in a list prefixed by a sign indicating whether it is normal (equal sign) or indexed (asterisk).

Outside the class definition, a number of accompanying functions are created automatically. (We hope you enjoy them!)

- A *predicate* recognizes objects of this class. Its name is made from the name of the class, suffixed as predicates usually are in Scheme with a question mark.
- One object *allocator* returns new instances of the class where the fields are not initialized. Their initial value can consequently be anything. Since the size of indexed fields must be specified during allocation, this allocator takes as many sizes as arguments as there are indexed fields in the class. The name of the allocator is made from the class name prefixed by `allocate-`.

- Another object *allocator* returns new instances of the class and specifies the initial values of their fields. The values constituting an indexed field are prefixed by the size of that field. The name of this allocator is made from the class name prefixed by `make-`.
- There are *selectors* for both normal and indexed fields. For each field of the class, there is a read-selector named by the class, a hyphen, and the field. The name of the corresponding write-selector is prefixed by `set-` and suffixed by the usual exclamation point, underlining the fact that it makes physical modifications. Selectors for indexed fields take a supplementary argument, an index.
- A function whose name is made from the name of the associated read-selector suffixed by `-length` accesses the size of each indexed field.

To support reflective operations, the class which is being created and which is itself an instance of the class `Class` is the value of the global variable that has the name of the class suffixed by `-class`. Here's an example of the descriptions of functions and variables[7] generated by `(define-class ColoredPolygon Point (color))`.

```
(ColoredPolygon? o)                                  → a Boolean
(allocate-ColoredPolygon sides-number)               → a polygon
(make-ColoredPolygon x y sides-number sides...color) → a polygon
(ColoredPolygon-x o)                                 → a value
(ColoredPolygon-y o)                                 → a value
(ColoredPolygon-side o index)                        → a value
(ColoredPolygon-color o)                             → a value
(set-ColoredPolygon-x! o value)                      → unspecified
(set-ColoredPolygon-y! o value)                      → unspecified
(set-ColoredPolygon-side! o value index)             → unspecified
(set-ColoredPolygon-color! o value)                  → unspecified
(ColoredPolygon-side-length o)                       → a length
ColoredPolygon-class                                 → a class
```

As we indicated, classes are created by `define-class`. That special form will be implemented by a macro, unfortunately with all the problems that implies. First among those problems is that `define-class` is a macro with an internal state—its inheritance hierarchy—but the system of Scheme macros, according to [CR91b], does not allow that sort of thing. For that reason, we're going to assume that we have another macro at our disposal, `define-meroonet-macro`, and its purpose is to define macros with no restrictions on the way they are expanded. We can code `define-meroonet-macro` in any existing implementation, but not in portable Scheme.

Is that internal state of `define-class` really necessary? Yes, in MEROONET, it is, for the following reason: when a class is defined, a number of functions are created to access its fields. For example, to define the class `Point` with the

7. We've arbitrarily adopted the convention of naming indexed fields by singular words. We think it's natural to write `(ColoredPolygon-side` *o i*`)` to get the i_{th} side of a polygon and to use `side` rather than `sides` in that context.

fields `x` and `y`, the read-accessors `Point-x` and `Point-y` are created. When we define the class `Polygon`, MEROONET does not impose `Point-x` and `Point-y` as the sole accessors for the fields `x` and `y`. Instead, MEROONET creates the functions `Polygon-x` and `Polygon-y`. The definition of the class `Polygon` does not mention inherited fields; the only way of knowing about them is through the name of the superclass. Therefore, the internal state that `define-class` maintains associates the names and types of its fields with each class.

We could avoid this inconvenience by adopting another convention for naming selectors. For example, we could omit the class name from the name of the selector. To illustrate that case, let's assume that the reader for the field `x` is named `get-x`. Then the definition of the class `Polygon` would modify the value of `get-x` to indicate how to extract the field `x` from a polygon. The most simple approach then is to make `get-x` a generic function which acquires new methods as classes are defined. This is the technique that CLOS uses where it is possible during the specification of a field to mention the generic function to which a method must be added to read that field. This decision makes generic functions mutable—a situation that might prove inconvenient for static optimizations during compilation because of the difficulty then of basing anything on a value changing without restrictions.

In contrast, we have decided to make selectors pure functions, not susceptible to modifications, so we can facilitate their integration inline. Of course, that implies some subtle problems, too, like the difference that might exist between `Point-x` and `Polygon-x`. Basically, both these functions extract the same field (`x`), but more precisely, the first one extracts `x` from a point, while the second gets it from a polygon. It seems normal then for the form `(Polygon-x (make-Point 11 22))` to raise an error and that in consequence, the selectors `Point-x` and `Polygon-x` should be different. Apart from this Byzantine situation, the greatest inconvenience of this decision is the large number of variables and global functions that it consumes. This may create a problem in computer memory but not in our own since all the names are formed systematically and the only reason for naming is for memory.

Recognizing the fields of the superclass (maintained by the internal state of `define-class`) comes into play not only in the way selectors are named: allocators need that information, too, so that their arity will be known and their compilation will be efficient. We'll get back to these ideas and elaborate them more in later sections.

In light of all these considerations, for MEROONET, we'll adopt the following solution: we define a class by constructing an object that is an instance of the class `Class` and inserting it in the hierarchy of classes. All that activity will be carried out by `register-class`. The functions accompanying that class are generated by the expansion function `Class-generate-related-names`. Conforming to the preceding description, we will temporarily define `define-class` like this:

```
(define-meroonet-macro (define-class name super-name
                                     own-field-descriptions )
  (let ((class (register-class name super-name own-field-descriptions)))
    (Class-generate-related-names class) ) )
```

That definition is problematic with respect to compilation since it mixes expansion time and evaluation time. The class is created and then inserted in the inheritance hierarchy during macro expansion, while the accompanying functions

are created during the evaluation of the expanded code. Let's assume that we compile a file containing the definition of the class **Polygon**. We can see the resulting code in a file with the suffix **.o** (as with compilers into C, like KCL [YH85], Scheme→C [Bar89], or Bigloo [Ser94]) or with a suffix like **.fasl** in a lot of other compilers. But the resulting code contains only the compilation of the expansion, that is, the definition of the accompanying functions. The class has really been created in the memory state of the compiler but not at all in the memory of the Lisp or Scheme system which will be linked to this **.o** file or which will dynamically load the **.fasl** file. The class has thus evaporated and disappeared by the time the compiler has finished its work.

Consequently, the construction of a class must appear in the expansion, but if it appears only there, then we can no longer require the class to generate the accompanying functions since we need to know the fields of the superclass during macro expansion to generate[8] all the selectors. Consequently, it is necessary to know the class at macro expansion and for it to appear in macro expansion. This dual existence makes it possible to compile the definitions of a class and its subclasses in the same file, as for example, **Point** and **Polygon**. To keep MEROONET from building the same class twice[9] during interpretation—once at macro expansion and then again at the evaluation of the expanded code—we will cleverly do this: when a class is defined during macro expansion, it will be stored in the global variable ***last-defined-class***; then at evaluation, the class will be created only if ***last-defined-class*** is empty.

```
(define *last-defined-class* #f)
(define-meroonet-macro (define-class name supername
                                     own-field-descriptions )
  (set! *last-defined-class* #f)
  (let ((class (register-class name supername own-field-descriptions)))
    (set! *last-defined-class* class)
    `(begin
       (if (not *last-defined-class*)
         (register-class ',name ',supername ',own-field-descriptions) )
      ,(Class-generate-related-names class) ) ) )
```

11.4 Other Problems

Still illustrating the difficulties of macros with real-life examples (namely, from MEROONET), we note that the order in which macro are expanded is important when we use macros with an internal state, as in **define-class**. Consider the following composed form in that context:

```
(begin (define-class Point Object (x y))
       (define-class Polygon Point ((* side))) )
```

If the expansion goes from left to right, then the class **Point** is defined and is thus available for the definition of the class **Polygon**. However, in the opposite case,

8. Remember that there is no linguistic means in Scheme nor at execution for creating a new global variable with a computed name.

9. We've made no effort in MEROONET to give a specific meaning to the redefinition of classes, and for that reason, we avoid redefining them.

the class **Polygon** can't be constructed because its superclass is not yet known. We could get around this problem by a (more complicated) way of deferring the construction of **Polygon** as long as the definition of **Point** has not yet been expanded. The definition of **Polygon** would then be empty, whereas the definition of **Point** would contain both.

So how do we compile the definition of **ColoredPolygon** in a file different from the one containing the definitions of **Point** and **Polygon**? We have to get the fields of its superclass somehow. MEROONET doesn't trouble itself about this problem: it simply requires all classes to be compiled together. MEROON, in contrast, adopts a different strategy. A class that will be inherited must be defined either before in the same file or with the keyword **:prototype**. That keyword[10] inserts the class in the class hierarchy at the right place without generating the equivalent code. Thus we'll write this:

```
(define-class Polygon Point ((* side)) :prototype)
(define-class ColoredPolygon Polygon (color))
```

Yet another solution is to use modules with an import/export mechanism to indicate which of those modules contain information pertinent to compile which others. That solution boils down to having a kind of data base, mimetically equivalent to the internal state of the compiler projected onto the file system.

11.5 Representing Classes

Classes in MEROONET are represented by objects of MEROONET. This representation makes self-description of the system easier. It also makes it easier to write metamethods based on the structure of classes, rather than on their inheritance. For example, how to migrate from machine to machine or how to print MEROONET objects by default can be deduced from their structure, that is, from their class. Consequently, there is a class—**Object**—defining all MEROONET objects. That class serves as the root for inheritance. Classes themselves are instances of the class **Class**. Field descriptors are instances of either **Mono-Field** or **Poly-Field**, both subclasses of **Field**.

When we expand the definition of a class, we make use of a number of functions accompanying these predefined classes, such functions as **Mono-Field?** or **make-Poly-Field**, etc. Should we deduce from that that when a class exists, we can use the functions accompanying it? More specifically, once **Point** has been macro expanded, can we use the function **make-Point**? If we take account of the preceding **define-class**, we must admit that we cannot use those functions because **make-Point** is not created by expansion but by evaluation. As a result, metaclasses pose subtle compilation problems—problems that MEROONET does not address.

As a minimum, we put only what is strictly necessary into a class: its name, the associated number, the list of its fields, its superclass, the list of numbers of its subclasses. That information about subclasses will be useful when we talk about

10. It's actually a little more complicated in MEROON. The option **:prototype** is expanded as a test that verifies at evaluation whether the prototype conforms to the real class which must exist then.

generic functions. We register the numbers rather than the subclasses themselves in order to avoid circularity at the cost of a slight decrease in efficiency. We can summarize all that information by saying that **Class**, the class of all classes, is defined like this:

```
(define-class Class Object
   (name number fields superclass subclass-numbers) )
```

Fields themselves are defined as objects in MEROONET. Fields are characterized by their type, their name, and the class that introduces them. In that latter field, useful for the general function **field-value**, we will store the number of the class, rather than the class itself so that classes and fields can be printed. Thus we have this:

```
(define-class Field Object (name defining-class-number))
(define-class Mono-Field Field ())
(define-class Poly-Field Field ())
```

Finally, the following function will give the illusion of extracting the class that introduced it from a field without bothering about the underlying number.

```
(define (Field-defining-class field)
  (number->class (careless-Field-defining-class-number field)) )
```

Of course, before MEROONET is installed, we can't define these classes but they are needed by MEROONET itself. To get around this bootstrapping problem, we create them by hand, like this:

```
(define Object-class
  (vector 1                                  ;it is a class
          'Object                            ;name
          0                                  ;class-number
          '()                                ;fields
          #f                                 ;no superclass
          '(1 2 3)                           ;subclass-numbers
          ) )
(define Class-class
  (vector
   1                                         ;it is also a class
   'Class                                    ;name
   1                                         ;class-number
   (list                                     ;fields
    (vector 4 'name             1)           ;offset= 1
    (vector 4 'number           1)           ;offset= 2
    (vector 4 'fields           1)           ;offset= 3
    (vector 4 'superclass       1)           ;offset= 4
    (vector 4 'subclass-numbers 1) )         ;offset= 5
   Object-class                              ;superclass
   '() ) )
(define Generic-class
  (vector
   1
   'Generic
   2
   (list
```

```
    (vector 4 'name           2)        ;offset= 1
    (vector 4 'default        2)        ;offset= 2
    (vector 4 'dispatch-table 2)        ;offset= 3
    (vector 4 'signature      2) )      ;offset= 4
    Object-class
    '() ) )
(define Field-class
  (vector
   1
   'Field
   3
   (list
    (vector 4 'name                  3) ;offset= 1
    (vector 4 'defining-class-number 3) ;offset= 2
    )
   Object-class
   '(4 5) ) )
(define Mono-Field-class
  (vector 1
          'Mono-Field
          4
          (careless-Class-fields Field-class)
          Field-class
          '() ) )
(define Poly-Field-class
  (vector 1
          'Poly-Field
          5
          (careless-Class-fields Field-class)
          Field-class
          '() ) )
```

Afterwards, the classes are installed as they should be, like this:

```
(vector-set! *classes* 0 Object-class)
(vector-set! *classes* 1 Class-class)
(vector-set! *classes* 2 Generic-class)
(vector-set! *classes* 3 Field-class)
(vector-set! *classes* 4 Mono-Field-class)
(vector-set! *classes* 5 Poly-Field-class)
(set! *class-number* 6)
```

Since MEROONET is managed by means of the class hierarchy, which itself is represented by MEROONET objects, several functions (such as the accessors `Class-number` or `Class-fields`) are needed even before they are constructed. For that purpose, we will introduce their equivalent with a name prefixed by `careless-`. That name is justified by their definition: they don't verify the nature of their argument, so MEROONET uses them only advisedly.

```
(define (careless-Class-name class)
  (vector-ref class 1) )
(define (careless-Class-number class)
  (vector-ref class 2) )
```

```
(define (careless-Class-fields class)
  (vector-ref class 3) )
(define (careless-Class-superclass class)
  (vector-ref class 4) )
(define (careless-Field-name field)
  (vector-ref field 1) )
(define (careless-Field-defining-class-number field)
  (vector-ref field 2) )
```

11.6 Accompanying Functions

For accompanying functions, we'll use a very practical utility, `symbol-concatenate`, to form new names.

```
(define (symbol-concatenate . names)
  (string->symbol (apply string-append (map symbol->string names))) )
```

To generate accompanying functions, we have a couple of choices. The first is to generate their equivalent code directly; the second, to generate a form that will be evaluated in an appropriate closure. With the second choice, we can factor the work and thus share the text of functions for which closures are constructed, but that choice prevents fine-tuning the optimizations because there will be more calls to computed functions (rather than statically known ones) and as a consequence, they cannot be inlined. To highlight the differences between those two choices, here is how the allocator for the class `Polygon` would be defined in those two cases:

```
(define allocate-Polygon
        (lambda (size)
          (let ((o (make-vector (+ 1 2 1 size))))
            (vector-set! o 0 (careless-Class-number Polygon-class))
            (vector-set! o 3 size)
            o )
(define allocate-Polygon (make-allocator Polygon-class))
```

The first choice lets us know statically that the allocator is a unary function and that its body uses only trivial functions, like `make-vector`, for reading and writing vectors. Consequently, it can be inlined readily in a few instructions and even compiled into a C macro, if we are compiling into C. In contrast, the second definition says nothing about arity. In fact, it doesn't even tell us[11] that this is a function that will be bound to the variable `allocate-Polygon`. Even so, we'll go with the second choice because the first one is more complicated to present and requires more memory for execution and compilation, so here is how accompanying functions are generated.

```
(define (Class-generate-related-names class)
  (let* ((name (Class-name class))
         (class-variable-name (symbol-concatenate name '-class))
         (predicate-name (symbol-concatenate name '?))
         (maker-name (symbol-concatenate 'make- name))
```

11. In Lisp, we can't even be sure that the computation terminates, and in Scheme, we don't know whether it returns a unique value.

```
      (allocator-name (symbol-concatenate 'allocate- name)) )
  `(begin
     (define ,class-variable-name (->Class ',name))
     (define ,predicate-name (make-predicate ,class-variable-name))
     (define ,maker-name (make-maker ,class-variable-name))
     (define ,allocator-name (make-allocator ,class-variable-name))
     ,@(map (lambda (field) (Field-generate-related-names field class))
            (Class-fields class) )
     ',(Class-name class) ) ) )
```

11.6.1 Predicates

For every class in MEROONET, there is an associated predicate that responds True about objects that are instances of that class, whether direct instances or instances of subclasses. The speed of this predicate is important because Scheme is a language without static typing so we are forever and again verifying the types or the classes of the objects we are handling. At compilation, we could try to guess[12] about the types of objects, factor the tests for types, make use of help from user declarations, or even impose a rule that programs must be well typed, for example, by defining methods that give information about their calling arguments.

The test about whether an object belongs to a class is thus very basic and depends on the general predicate **is-a?**. To keep up its speed, **is-a?** assumes that the object is an object and that the class is a class as well. Consequently, we can't apply **is-a?** to just anything. In contrast, the predicate associated with a class is less restricted. It will first test whether its argument is actually a MEROONET object. Finally, we'll introduce a last predicate to use for its effect in verifying membership and issuing an intelligible message by default. Since errors are not standard in Scheme,[13] any errors detected by MEROONET call the function **meroonet-error**, which is not defined—one portable way of getting an error!

```
(define (make-predicate class)
  (lambda (o) (and (Object? o)
                   (is-a? o class) )) )
(define (is-a? o class)
  (let up ((c (object->class o)))
    (or (eq? class c)
        (let ((sc (careless-Class-superclass c)))
          (and sc (up sc)) ) ) ) )
(define (check-class-membership o class)
  (if (not (is-a? o class))
    (meroonet-error "Wrong class" o class) ) )
```

The complexity of the predicate **is-a?** is linear, since we test whether the object belongs to the target class, or whether the superclass is the target class, or whether the superclass of the superclass is the target and so on. This strategy is not too bad because the first try is usually right, at least in roughly one try out of two.

12. The form **define-method** indicates the class of the discriminating variable, and we could take advantage of that hint.

13. At least at the moment we're writing, under the reign of R^{4}RS [CR91b].

However, there is a potential problem of infinite regression in `is-a?`. To get the superclass, we use the function `careless-Class-superclass`, rather than the function `Class-superclass` directly. The difference between those two functions is that the careless one assumes that its first argument is a class. That assumption is safe enough in the context of `is-a?`. If we had used the other function, it would have tested whether its argument was really a class and in doing so, it would have recursively required the predicate `is-a?` and thus seriously impaired the efficiency of MEROONET.

11.6.2 Allocator without Initialization

MEROONET offers two kinds of allocators. The first creates objects whose contents may be anything; the second creates objects all of whose fields have been initialized. The same concepts exist in Lisp and in Scheme: `cons` is the allocator with initialization for dotted pairs; `vector` is the allocator with initialization for vectors. There is also a second allocator for vectors: without a second argument, `make-vector` creates vectors with unspecified contents. MEROONET supports both types of allocation.

Allocation without initialization means that the contents of an uninitialized object might be anything. There are at least two ways to understand that idea. For reasons having to do with garbage collection, (and except for garbage collectors that tolerate certain ambiguities as in [BW88]), uninitialized means generally that the object is filled with entities known to the implementation. There are two different possible semantics here, depending on whether or not those entities are first class values.

- When the entity `#<uninitialized>` appears in a field of a data structure, it explicitly marks the field as containing no value. Attempts to read such a field raise an error since there is no value associated with it.
- In implementations where there is no such entity, fields are initialized with normal values, but they might be anything. For example, in Lisp, it's often `nil`; in Le-Lisp, it's `t`; in many Scheme implementations, it's `#f`. The initial content of a field is consequently undefined, that is, unspecified, in short, anything, but it is not an error to read such a field.

There is a third interpretation—one we call "C style"—in which reading an uninitialized field has unpredictable consequences. For that reason, we strongly urge you not to try reading such a field. This interpretation is even less specific than the previous ones. For example, in this interpretation, it is not obligatory for the implementation to detect an error when there is an attempt to read such a field. Of course, since this read-procedure doesn't carry out any tests on the value it reads, it is very fast! However, when we attempt to read such a field, we may very well get the infamous but informative message "`Bus errror, core dumped`" or even worse, some result that has nothing to do with what we're trying to compute but which we might take as valid, given no warning. The proof that we never (truly never!) read an uninitialized field is left to the programmer.

The first interpretation—with the entity `#<uninitialized>`—obliges a reader to raise an error when it encounters an uninitialized field. Obviously, that costs

an explicit test every time a field that might be uninitialized is read, according to [Que93b]. The second interpretation does not require such a test, so it compares favorably in this respect with C.

In terms of allocation, the C-style interpretation is even more efficient because there is no need to initialize fields nor to clear them. Paradoxically, allocation with explicit initialization is usually more efficient than allocation without initialization. In practice, allocating without initialization on the part of the user does in fact require an initialization—with `#<uninitialized>`—at a cost comparable to initialization with any value at all, like `#t`, for example. But most of the time, a field that is not initialized when it is allocated will be initialized right away anyway (and that definitely doubles the work), whereas allocation with explicit initialization fills all the constant fields definitively in one fell swoop.

According to that first interpretation, CLOS guarantees that it detects an uninitialized field. In Scheme, it's helpful to realize that variables are in fact represented by entities that can be defined in MEROONET and for which the detection of uninitialized fields is obligatory. **[see p. 60]** Since we have decided to ignore the formalists, we don't require MEROONET to handle uninitialized fields, and thus we simplify the code: uninitialized fields will be filled with `#f`.

To be useful, the idea of an allocator without initialization means that a created object must be mutable. This point is important because immutable objects are naturally closer to mathematics than are objects that have an internal state, especially so with respect to equality. **[see p. 122]** Immutable objects lend themselves well to optimization since their contents are guaranteed not to vary. Likewise, in the function `make-allocator` that we looked at earlier, we would be able to precompute the expression `(Class-number class)` (just as we do `(Class-fields class)`) if the field `number` in the `Class` were immutable. In that way, we could directly record its value rather than have to recompute it for every allocation.

An allocator without explicit initialization will be returned by the function `make-allocator`. It will take a class and return a function that accepts as arguments a list of natural integers specifying the size of each possible indexed field appearing in the definition of the class. The first computation consists of determining the size of the zone in memory (of the vector) to reserve (to allocate); to do that, we recursively run through the list of fields and their sizes. How do we know the list of fields? We extract it from the class by `Class-fields`. Once the zone in memory has been reserved, we must structure it to put the size of indexed fields there. That's the purpose of the second loop running over the fields and their sizes and maintaining a current offset inside the allocated memory zone. Finally, the object, having just acquired its "skeleton" and the instantiation link to the class where it belongs, is returned as a value.

```
(define (make-allocator class)
  (let ((fields (Class-fields class)))
    (lambda sizes
      ;;compute the size of the instance to allocate
      (let ((room (let iter ((fields fields)
                             (sizes sizes)
                             (room *starting-offset*) )
                    (if (pair? fields)
```

```
                    (cond ((Mono-Field? (car fields))
                           (iter (cdr fields) sizes (+ 1 room)) )
                          ((Poly-Field? (car fields))
                           (iter (cdr fields) (cdr sizes)
                                 (+ 1 (car sizes) room) ) ) )
                    room ) )))
  (let ((o (make-vector room #f)))
    ;; setup the instantiation link and skeleton of the instance
    (vector-set! o 0 (Class-number class))
    (let iter ((fields fields)
               (sizes sizes)
               (offset *starting-offset*) )
      (if (pair? fields)
          (cond ((Mono-Field? (car fields))
                 (iter (cdr fields) sizes (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (vector-set! o offset (car sizes))
                 (iter (cdr fields) (cdr sizes)
                       (+ 1 (car sizes) offset) ) ) )
          o ) ) ) ) ) ) )
```

We have a few remarks to make about that code.

1. The allocators that are created take a list of sizes as their argument. Consequently, they consume dotted pairs if they are associated with classes that have at least one indexed field. To be able to allocate, we are obliged to allocate that much! Later, we'll sketch a solution to this problem.
2. Superfluous sizes are simply ignored without generating an errror.
3. Allocation, as a procedure, runs over the list of fields in the class twice. Doing so twice is costly, especially if the list doesn't vary: we know once the class has been created whether the direct instances of the class have a fixed size or not. We'll correct this point later, too.
4. If a size is not a natural number (thus non-negative), addition will lead to a computation of the size of the memory zone to allocate that will be in error, but the error message will be very cryptic. To be more "user-friendly" in this respect, an explicit test must be carried out, and a relevant and intelligible error message should be generated.
5. Fields can be instances only of `Mono-Field` or `Poly-Field`. It's not possible to add user-defined classes of fields here.

11.6.3 Allocator with Initialization

Allocators with initialization are created in a similar way, but as we go, we'll improve the efficiency of allocators for objects of small fixed size. In Scheme, allocators without initialization (like `make-vector` or `make-string`) take the size of the object to allocate, followed by a possible value to fill in. That value will be used to initialize all components. Allocators with initialization (like `cons`, `vector`, or `string`) successively take all the initialization values. The number of initialization values becomes the size of a unique indexed field for `vector` and `string`. Since

we allow multiple indexed fields simultaneously, we must know the size for each of them since we cannot infer it. Consequently, MEROONET requires the initialization values of indexed fields to be prefixed by their number. We could allocate a three-sided polygon by writing this:

```
(make-ColoredPolygon 'x 'y 3 'Point0 'Point1 'Point2 'color)
```

For allocation with intialization, we'll adopt the following technique: all the arguments are gathered into one list, **parms**; a vector is allocated with all these values and with the number of the appropriate class as its first component. Then all we have to do is to verify that the object has the correct structure for MEROONET, so we run through the arguments and the fields of its class to verify whether there are at least as many arguments as necessary. After this verification, the object is returned. This technique is simpler (and thus might be faster) than the one earlier that seemed more natural, the one consisting of verifying all the arguments, then allocating only after the object to be returned.

```
(define (make-maker class)
  (or (make-fix-maker class)
      (let ((fields (Class-fields class)))
        (lambda parms
          ;;create the instance
          (let ((o (apply vector (Class-number class) parms)))
            ;;check its skeleton
            (let check ((fields fields)
                        (parms parms)
                        (offset *starting-offset*) )
              (if (pair? fields)
                  (cond ((Mono-Field? (car fields))
                         (check (cdr fields) (cdr parms) (+ 1 offset)) )
                        ((Poly-Field? (car fields))
                         (check (cdr fields)
                                (list-tail (cdr parms) (car parms))
                                (+ 1 (car parms) offset) ) ) )
                  o ) ) ) ) ) ) )
```

Well, this allocator ignores superfluous arguments, but it is still not very efficient because it allocates the arguments in a list that it then converts into a vector. Since this list (the value of **parms**) is used to verify the structure of the object rather than to verify the components of the object directly, we can't even hope for a compiler sufficiently intelligent not to create that list of arguments. For that reason, we'll stick a little wart on the nose of **make-maker** to improve the case of allocators for objects that have no indexed fields.

```
(define (make-fix-maker class)
  (define (static-size? fields)
    (if (pair? fields)
        (and (Mono-Field? (car fields))
             (static-size? (cdr fields)) )
        #t ) )
  (let ((fields (Class-fields class)))
    (and (static-size? fields)
         (let ((size (length fields))
```

```
        (cn (Class-number class)) )
    (case size
      ((0)  (lambda () (vector cn)))
      ((1)  (lambda (a) (vector cn a)))
      ((2)  (lambda (a b) (vector cn a b)))
      ((3)  (lambda (a b c) (vector cn a b c)))
      ((4)  (lambda (a b c d) (vector cn a b c d)))
      ((5)  (lambda (a b c d e) (vector cn a b c d e)))
      ((6)  (lambda (a b c d e f) (vector cn a b c d e f)))
      ((7)  (lambda (a b c d e f g) (vector cn a b c d e f g)))
      (else #f) ) ) ) ) )
```

So allocators of classes without any indexed fields and with fewer than nine normal fields are produced by closures with fixed arity. If a class has no indexed fields but more than nine normal fields, we go back to the preceding case, which verifies whether the number of initialization values is sufficient. Always hoping to minimize error detection, we'll make `cdr` or `list-tail` detect the fact when there are not enough initialization values. Those two must not truncate a list that is already empty. In the case where we're allocating small-sized objects, we'll rely on the arity check. Of course, since we are relying on various strategies to detect the same anomaly, we'll get error messages about it that are not uniform—a situation that is not the best, but the code for MEROONET would balloon by more than a fourth if we tried to be more "user-friendly" in this respect.

A native implementation of MEROONET must support efficient allocation of objects. To avoid that monstrous situation we saw earlier of allocating in order to allocate, it should probably be endowed with an internal means of building allocators of the type `(vector cn a b c ... )`.

11.6.4 Accessing Fields

For every field of a class, MEROONET creates accompanying functions to read, to write, and to get the length of a field, if it is indexed. The accompanying functions related to fields (the selectors) are constructed by the subfunction of macro expansion: `Field-generate-related-names`.

```
(define (Field-generate-related-names field class)
  (let* ((fname (careless-Field-name field))
         (cname (Class-name class))
         (cname-variable (symbol-concatenate cname '-class))
         (reader-name (symbol-concatenate cname '- fname))
         (writer-name (symbol-concatenate 'set- cname '- fname '!)) )
    `(begin
       (define ,reader-name
         (make-reader
          (retrieve-named-field ,cname-variable ',fname) ) )
       (define ,writer-name
         (make-writer
          (retrieve-named-field ,cname-variable ',fname) ) )
       ,@(if (Poly-Field? field)
             `((define ,(symbol-concatenate cname '- fname '-length)
                 (make-lengther
```

```
                (retrieve-named-field ,cname-variable ',fname) ) ))
            '() ) ) ) )
```

As before, the accompanying functions are constructed by closure, not by code synthesis. The constructors are **make-reader**, **make-writer**, and **make-lengther**. As arguments, they all take a field and a class. Since the constructors have to be built at evaluation, they find these objects by the name of the global variable that contains the class and by the function **retrieve-named-field**, which gets a field in a class by name.

```
(define (retrieve-named-field class name)
  (let search ((fields (careless-Class-fields class)))
    (and (pair? fields)
         (if (eq? name (careless-Field-name (car fields)))
             (car fields)
             (search (cdr fields)) ) ) ) )
```

11.6.5 Accessors for Reading Fields

Since we have two types of fields—indexed and normal—we also have two kinds of readers with different arity. In case the implementation is not to your taste, we should point out that a constant offset accesses fields not preceded by an indexed field. The function **make-reader** for constructing readers, of course, has to take this fact into account. We'll run through the list of fields in order to determine whether that offset is constant or not. If it is, an appropriate function will be generated; otherwise, we'll resort to a general function for reading fields: **field-value**. In that way, all the fields with a constant offset (possibly indexed) are read efficiently.

Readers are safe functions in the sense that they verify whether the object to which they are applied belongs to the right class, that is, a class inheriting from the class that introduced the field in the first place. That's the role of the function **check-class-membership**, that we looked at earlier. Another reason we speak of them as safe functions is that the reader of an indexed field uses the function **check-index-range** to test whether the index is correct with respect to the size of the indexed field.

```
(define (make-reader field)
  (let ((class (Field-defining-class field)))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          (cond ((Mono-Field? (car fields))
                 (lambda (o)
                   (check-class-membership o class)
                   (vector-ref o offset) ) )
                ((Poly-Field? (car fields))
                 (lambda (o i)
                   (check-class-membership o class)
                   (check-index-range i o offset)
                   (vector-ref o (+ offset 1 i)) ) ) )
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
```

```
                ((Poly-Field? (car fields))
                 (cond ((Mono-Field? field)
                        (lambda (o)
                          (field-value o field) ) )
                       ((Poly-Field? field)
                        (lambda (o i)
                          (field-value o field i) ) ) ) ) ) ) ) ) )
(define (check-index-range i o offset)
  (let ((size (vector-ref o offset)))
    (if (not (and (>= i 0) (< i size)))
      (meroonet-error "Out of range index" i size) ) ) )
```

For fields that are located after an indexed field, we'll make do with a mechanism that incarnates the general (not generic) function **field-value** because MEROONET does not support the creation of new types of fields. The function **field-value**, of course, is associated with the function **set-field-value!**. Computations about offsets common to both those functions are concentrated in **compute-field-offset**.

```
(define (compute-field-offset o field)
  (let ((class (Field-defining-class field)))
    ;;(assume (check-class-membership o class))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          offset
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (skip (cdr fields)
                       (+ 1 offset (vector-ref o offset)) ) ) ) ) ) ) )
(define (field-value o field . i)
  (let ((class (Field-defining-class field)))
    (check-class-membership o class)
    (let ((fields (careless-Class-fields class))
          (offset (compute-field-offset o field)) )
      (cond ((Mono-Field? field)
             (vector-ref o offset) )
            ((Poly-Field? field)
             (check-index-range (car i) o offset)
             (vector-ref o (+ offset 1 (car i))) ) ) ) ) )
```

11.6.6 Accessors for Writing Fields

The definition of an accessor to write a field is analogous to that of a reader so it poses no difficulties. In contrast, the function **set-field-value!** has a strange signature. The signature of a writer for fields is based on the signature for a reader with the value to add at the tail of the arguments. That's the usual pattern in Scheme, for example, in **car** and **set-car!**. However, the fact that there may be

an optional index present upsets this nice pattern, so we've chosen this order[14]: `(o v field . i)`.

```
(define (make-writer field)
  (let ((class (Field-defining-class field)))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          (cond ((Mono-Field? (car fields))
                 (lambda (o v)
                   (check-class-membership o class)
                   (vector-set! o offset v) ) )
                ((Poly-Field? (car fields))
                 (lambda (o i v)
                   (check-class-membership o class)
                   (check-index-range i o offset)
                   (vector-set! o (+ offset 1 i) v) ) ) )
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (cond ((Mono-Field? field)
                        (lambda (o v)
                          (set-field-value! o v field) ) )
                       ((Poly-Field? field)
                        (lambda (o i v)
                          (set-field-value! o v field i) ) ) ) )
                ) ) ) ) ) )
(define (set-field-value! o v field . i)
  (let ((class (Field-defining-class field)))
    (check-class-membership o class)
    (let ((fields (careless-Class-fields class))
          (offset (compute-field-offset o field)) )
      (cond ((Mono-Field? field)
             (vector-set! o offset v) )
            ((Poly-Field? field)
             (check-index-range (car i) o offset)
             (vector-set! o (+ offset 1 (car i)) v) ) ) ) ) )
```

From a stylistic point of view, writers for fields are constructed with the prefix `set-`, as in `set-cdr!`. We could have used the suffix `-set!` just as well, as in `vector-set!`, but we chose the prefix to show as soon as possible that we are dealing with a modification. Visually, this choice also looks more like an assignment.

11.6.7 Accessors for Length of Fields

We can access the length of a field either by a specialized accompanying function or by the general function `field-length`. Here again, we've been able to improve access to any field that is not located after an indexed field.

```
(define (make-lengther field)
  ;;(assume (Poly-Field? field))
```

14. This order may remind you of putprop if you are already nostalgic about property lists.

```
  (let ((class (Field-defining-class field)))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          (lambda (o)
            (check-class-membership o class)
            (vector-ref o offset) )
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (lambda (o) (field-length o field)) ) ) ) ) ) )
(define (field-length o field)
  (let* ((class (Field-defining-class field))
         (fields (careless-Class-fields class))
         (offset (compute-field-offset o field)) )
    (check-class-membership o class)
    (vector-ref o offset) ) )
```

11.7 Creating Classes

The form **define-class** does not actually create the object to represent the class being defined. Rather, it delegates that task to the function **register-class**. That function actually allocates the object and then calls **Class-initialize!** to analyze the parameters of the definition of the class as well as to initialize the various fields of the class. Once the fields have been filled in (with the help of **parse-fields** for fields belonging to the class), the class is inserted in the class hierarchy. Finally, the call to **update-generics** will confer all the methods that the newly created class inherits from its superclass.

```
(define (register-class name supername own-field-descriptions)
  (Class-initialize! (allocate-Class)
                     name
                     (->Class supername)
                     own-field-descriptions ) )
(define (Class-initialize! class name superclass own-field-descriptions)
  (set-Class-number! class *class-number*)
  (set-Class-name! class name)
  (set-Class-superclass! class superclass)
  (set-Class-subclass-numbers! class '())
  (set-Class-fields!
   class (append (Class-fields superclass)
                 (parse-fields class own-field-descriptions) ) )
  ;;install definitely the class
  (set-Class-subclass-numbers!
   superclass
   (cons *class-number* (Class-subclass-numbers superclass)) )
  (vector-set! *classes* *class-number* class)
  (set! *class-number* (+ 1 *class-number*))
  ;;propagate the methods of the super to the fresh class
  (update-generics class)
```

```
  class )
```

The specifications of fields belonging to the class are analyzed by the function **parse-fields**. That function analyzes the syntax of these specifications. When a specification is parenthesized, it can begin only with an equal sign or an asterisk. Any other object in that position raises an error indicated by **meroonet-error**. MEROONET does not support the redefinition of an inherited field; the function **check-conflicting-name** verifies that point. In contrast, it does not check whether the same name appears more than once among the fields of the object.

```
(define (parse-fields class own-field-descriptions)
  (define (Field-initialize! field name)
    (check-conflicting-name class name)
    (set-Field-name! field name)
    (set-Field-defining-class-number! field (Class-number class))
    field )
  (define (parse-Mono-Field name)
    (Field-initialize! (allocate-Mono-Field) name) )
  (define (parse-Poly-Field name)
    (Field-initialize! (allocate-Poly-Field) name) )
  (if (pair? own-field-descriptions)
      (cons (cond
             ((symbol? (car own-field-descriptions))
              (parse-Mono-Field (car own-field-descriptions)) )
             ((pair? (car own-field-descriptions))
              (case (caar own-field-descriptions)
                ((=) (parse-Mono-Field
                      (cadr (car own-field-descriptions)) ))
                ((*) (parse-Poly-Field
                      (cadr (car own-field-descriptions)) ))
                (else (meroonet-error
                       "Erroneous field specification"
                       (car own-field-descriptions) )) ) ) )
            (parse-fields class (cdr own-field-descriptions)) )
      '() ) )
(define (check-conflicting-name class fname)
  (let check ((fields (careless-Class-fields (Class-superclass class))))
    (if (pair? fields)
        (if (eq? (careless-Field-name (car fields)) fname)
            (meroonet-error "Duplicated field name" fname)
            (check (cdr fields)) )
        #t ) ) )
```

11.8 Predefined Accompanying Functions

At this point, we've already presented the entire backbone for defining classes, but it is not yet completely operational. Indeed, we can't yet define a class because the predefined accompanying functions, like **Class**, **Field**, etc., haven't been covered yet. Well, we can't use **define-class** because those functions are missing, but the role of **define-class** is to define them, so we find ourselves stuck again with a

bootstrapping problem.

With a little patience, we'll define those functions "by hand" ourselves. We'll write just the ones that are minimally necessary and we'll even simplify the code by leaving out any verifications. If everything goes well, then we will expand the definitions of predefined classes and cut-and-paste that code, as we have done before. The truth, the whole truth, and nothing but the truth, is that we have to define the predicates before the readers, the readers before the writers, and the writers before the allocators. Foremost, we have to make the accompanying functions for `Class` appear before everything else. Here, we'll show you just a synopsis of these definitions because the entire thing would be a little monotonous.

```
(define Class? (make-predicate Class-class))
(define Generic? (make-predicate Generic-class))
(define Field? (make-predicate Field-class))
(define Class-name
  (make-reader (retrieve-named-field Class-class 'name)))
(define set-Class-name!
  (make-writer (retrieve-named-field Class-class 'name)))
(define Class-number
  (make-reader (retrieve-named-field Class-class 'number)))
(define set-Class-subclass-numbers!
  (make-writer (retrieve-named-field Class-class 'subclass-numbers)))
(define make-Class (make-maker Class-class))
(define allocate-Class (make-allocator Class-class))
(define Generic-name
  (make-reader (retrieve-named-field Generic-class 'name)))
(define allocate-Poly-Field (make-allocator Poly-Field-class))
```

At this stage now, it's possible to use `define-class`. Careful: don't abuse it to redefine the predefined classes `Object`, `Class`, and all the rest. For one thing, you mustn't do so because `define-class` is not idempotent[15] so you would get twelve initial classes, among which six would be inaccessible. For another, when you compiled, you would duplicate all the accompanying functions.

11.9 Generic Functions

Generic functions are the result of adapting the idea of sending messages—important in the object world—to the functional world of Lisp. Sending a message in Smalltalk [GR83] looks like this:

```
receiver message: arguments ...
```

As people often say, everything already exists in Lisp—all you have to do is add parentheses! The first importations of message sending in Lisp used a keyword, like `send` or `=>` as in Planner [HS75], to get this:

```
(send receiver message arguments ... )
```

15. One work-around would be to reset `*class-number*` to zero and redefine the classes which should have the same numbers.

The receiver, of course, is the object that gets the message. It is unique in Smalltalk; that is, you can't send a message to more than one object at a time. Now, in our way of thinking that anything can be an object in Lisp, we see right away that a function like binary addition makes sense only if it knows the class of both its objects. For two integers, that is, we use binary integer addition; for two floating-point numbers, we use binary floating-point addition; and for mixed cases, we must first convert the integer argument into a floating-point number before we add them (a practice known as "floating-point contagion"). In fact, addition has methods for which there is not a unique receiver. Such methods are known as *multimethods*. The syntax for sending messages puts the receiver in a privileged position and thus undermines the idea of multimethods, so Common Loops [BKK+86] proposed changing the syntax to something more suggestive, like this:

```
(message receiver arguments ... )
```

The keyword **send** has disappeared, and the message itself appears in the function position, so it's a function. Since it inspects its arguments in order to determine which method to apply, we say it is a *generic* function. A generic function takes into account all the arguments it receives, so there are certain consequences:

1. It can treat the ones it wants to as discriminants.
2. It might not give any particular pre-eminence to the first argument (the former receiver) among the others.
3. It can take more than one argument into account as a discriminant (for example, in the case of binary addition).

In short, the idea of generic functions generalizes the idea of message-sending, and it's this idea of generic functions that MEROONET offers. However, multimethods simply don't appear in this book. Moreover, they generally represent less than 5% of the cases in use, according to [KR90], so MEROONET doesn't even implement multimethods.

Some people interested in generic functions ask whether this generalization is still faithful to the spirit of objects. We won't attempt to respond to this delicate question. However, we will admit that generic functions completely hide the object-aspect since they are real functions that don't need a special operator like **send**. Whether or not a function should be generic and use message-sending is thus left as an implementation question and has no impact on its calling interface. On the implementation side, there is a slight surcharge due to encapsulation since the application of a generic function with a discriminating first argument (*g arguments* ...) shows that *g* is more or less equivalent to `(lambda args (apply send (car args) g (cdr args)))`.

How should we represent generic functions in Scheme? Since we have to be able to apply them, in Scheme they are necessarily represented by functions. However, since we want to preserve self-description, we would like for them to be MEROONET objects belonging to the class **Generic**, so we would like for them to be simultaneously objects *and* functions, in short, functional objects, or even *funcallable objects*. CLOS and OakLisp [LP86, LP88] support such concepts.

In fact, generic functions are really objects endowed with an internal state corresponding to the set of methods that they know. Consequently, we'll adopt the

following technique, even though it is largely suboptimal. A generic function will be represented simultaneously by a MEROONET object and a Scheme function, the value of the variable with the same name as the generic function. That function will contain the MEROONET object in its closure. Methods will be added to the object which will thus be visible to the Scheme function that will be applied. The generic object will be retrieved by its name (and that fact means that we cannot have anonymous generic functions) rather than by the value of the global variable bearing the same name; it will be retrieved in such a way as to be insensitive to the lexical context where generic functions and methods are defined.

The definition form for generic functions uses the following syntax:

`(define-generic (`*name variables ...* `) [` *default...* `] )`

The first term defines the form of the call to the generic function and specifies the discriminating variable (the receiver in Smalltalk). The discriminating variable appears between parentheses. It is possible to define the default body of the generic function in the rest of the form. In a language where all values were MEROONET objects, that would be equivalent to putting a method in the class of all values, namely, `Object`. Since, in this implementation of Scheme, there are values that are not objects, the default body will be applied to every discriminating value which is not a MEROONET object. That property makes the integration of MEROONET and Scheme simpler; it also supports customized error trapping. One of the novelties of MEROONET is that generic functions may have any signature, even a dotted variable. In any case, the methods must have a compatible signature.

The class `Generic` will be defined like this:

```
(define-class Generic Object (name default dispatch-table signature))
```

We can retrieve the generic function by name with `->Generic`. The field `default` contains the default body, either supplied by the user or synthesized automatically by default to raise an error. The signature makes it possible to judge the compatibility of methods that might be added to the generic function. This test is important, for one thing, because it would be insane to add methods of different arity: the call to a generic function is already varied enough in terms of the method that will be chosen without adding the problems presented by varying arity. For another thing, a native implementation could take advantage of the uniformity among methods to improve the call for generic functions, as in [KR90].

The internal state of a generic function is produced entirely by a vector indexed by the class numbers. The vector contains all the known methods of the generic function. This vector is known as the *dispatch table* for the generic function. Consequently, the means of calling a generic function is straightforward: the number of the class of the discriminating argument will be the index into the dispatch table to retrieve the appropriate method. This way of coding is extremely fast, but it takes up space since the set of these tables is equivalent to an $n * m$ matrix where n is the number of classes and m the total number of generic functions. There are techniques for compressing a dispatch table, as in [VH94, Que93b]. Those techniques are feasible mainly because most of the time the methods of a generic function involve only a subtree in the class hierarchy.

The methods of a generic function as well as its default body will all have the same finite arity, and that's also the case for generic functions with a dotted

variable. For example, let's assume that the generic function `f` is defined like this:

```
(define-generic (f a (b) . c) (g b a c))
```

Then the default body will be this:

```
(lambda (a b c) (g b a c))
```

And all the methods will be represented by functions with arity comparable to `(a b c)`. The arity of the image of a generic function in Scheme is the same arity specified for the generic function. For the preceding example, that will be:

```
(lambda (a b . c) ((determine-method G127 b) a b c))
```

The value of the variable `G127`[16] is the generic object containing the dispatch table from which the function **determine-method** chooses the appropriate method. Since not all values of Scheme are objects, we must first test whether the value of the discriminating variable is really a MEROONET object.

```
(define (determine-method generic o)
  (if (Object? o)
      (vector-ref (Generic-dispatch-table generic)
                  (vector-ref o 0) )
      (Generic-default generic) ) )
```

So here is the definition of generic functions, finally:

```
(define-meroonet-macro (define-generic call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((generic (gensym)))          ;make generic hygienic
       `(define ,(car call)
          (let ((,generic (register-generic
                           ',(car call)
                           (lambda ,(flat-variables variables)
                             ,(if (pair? body)
                                  `(begin . ,body)
                                  `(meroonet-error
                                    "No method" ',(car call)
                                    . ,(flat-variables variables) ) ) )
                           ',(cdr call) )))
            (lambda ,variables
              ((determine-method ,generic ,(car discriminant))
               . ,(flat-variables variables) ) ) ) ) ) ) ) )
```

The function **parse-variable-specifications** analyzes the list specifying the variables in order to extract the discriminating variable and to re-organize the list of variables to make it conform to what Scheme expects. It will be common to **define-generic** and **define-method**. It invokes its second argument on these two results. Nonchalantly not caring a bit about dogmatism, the function **parse-variable-specifications** does not verify whether there is only one discriminating variable.

16. To make the expansion of **define-generic** cleaner, the variable enclosing the generic object has a name synthesized by **gensym**.

```
(define (parse-variable-specifications specifications k)
  (if (pair? specifications)
      (parse-variable-specifications
       (cdr specifications)
       (lambda (discriminant variables)
         (if (pair? (car specifications))
             (k (car specifications)
                (cons (caar specifications) variables) )
             (k discriminant (cons (car specifications) variables)) ) ) )
      (k #f specifications) ) )
```

The default body of the generic function is built then. A generic object is constructed by **register-generic** making it possible (at the level of the results of expanding of **define-generic**) to hide the coding details of generic functions, especially the variable ***generics***. Among these details, the dispatch table is constructed, so we allocate a vector that we initially fill with the default body. That default body is effectively the method which will be found if there isn't any other method around. The size of the dispatch table depends on the total number of possible classes, not on the number of actual classes. This fact means that when new classes are defined, we have to increase the size of the dispatch table for all existing generic functions.

```
(define (register-generic generic-name default signature)
  (let* ((dispatch-table (make-vector *maximal-number-of-classes*
                                      default ))
         (generic (make-Generic generic-name
                                default
                                dispatch-table
                                signature )) )
    (set! *generics* (cons generic *generics*))
    generic ) )
```

The function **flat-variables** flattens a list of variables and transforms any final possible dotted variable into a normal one. That transformation occurs in the default body but will also be applied to any methods to come.

```
(define (flat-variables variables)
  (if (pair? variables)
      (cons (car variables) (flat-variables (cdr variables)))
      (if (null? variables) variables (list variables)) ) )
```

A Bit More about Class Definitions

We've already mentioned the role of the function **update-generics** when a class is defined. Its role is to propagate the methods of the superclass to this new class. If we first define the class **Point** with the method **show** to display points, and then we define the class **ColoredPoint**, it seems normal for **ColoredPoint** to inherit that method. To implement that characteristic, we need to have all the generic functions available so that we can update them.

```
(define (update-generics class)
  (let ((superclass (Class-superclass class)))
```

```
(for-each (lambda (generic)
            (vector-set! (Generic-dispatch-table generic)
                         (Class-number class)
                         (vector-ref (Generic-dispatch-table generic)
                                     (Class-number superclass) ) ) )
          *generics* ) ) )
```

11.10 Method

Methods are defined by **define-method** of course. Its syntax resembles the syntax of **define**. The list of variables is similar to the one that appears in **define-generic**: the discriminating variable occurs between parentheses as does the name of the class for which the method is installed.

(**define-method** (*name variables ...*) *forms ...*)

The generic functions that we've already defined are objects on which we can confer new behavior or methods by means of **define-method**. Thus in MEROONET, generic functions are mutable objects, and to that degree, they make optimizations difficult, unless we freeze the class hierarchy (as with **seal** in Dylan [App92b]) and then analyze the types (or rather, the classes) a little. Making generic functions immutable would be another solution, but that requires functional objects, which we've excluded here.

We have already evoked the arity of methods for the default body in the definition of the functional image for generic functions in Scheme. Once the method has been constructed, it must be inserted in the dispatch table not only for the class for which it is defined but also for all the subclasses for which it is not redefined.

The remaining problem is the implementation of what Smalltalk calls **super** and what CLOS calls **call-next-method**; that is, the possibility for a method to invoke the method which would have been invoked if it hadn't been there. The form (**call-next-method**) can appear only in a method and corresponds to invoking the supermethod with implicitly the same[17] arguments as the method. Of course, the supermethod of the class **Object** is the default body of the generic function.

The way that the function **call-next-method** local to the body of the method searches for the supermethod is inspired by **determine-method**, but this time we need to test whether the value of the discriminating variable is indeed an object and whether the number of the class to consider for indexing is no longer the number of the direct class of the discriminating value but rather that of the superclass. Thus in order to install a method that uses **call-next-method**, we have to know the number of the superclass of the class for which we are installing the method. Careful: because of special considerations in macro expansion, this number should not be known right away but only at evaluation. For that reason, we have to resort to the following technique. The form **define-method** will build a premethod, that is, a method using as parameters the generic function and class where it will be

17. We haven't kept the possibility of changing these arguments as in CLOS because it would then be possible to change the contents of the discriminating variable to anything at all, and that would violate the intention of the supermethod as well as the hypotheses of the compilation, probably.

installed. In order to hide the details of this installation and to reduce the size of the macro expansion of `define-method`, we'll create the function `register-method`, like this:

```
(define-meroonet-macro (define-method call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((g (gensym))(c (gensym)))    ;make g and c hygienic
       `(register-method
         ',(car call)
         (lambda (,g ,c)
           (lambda ,(flat-variables variables)
             (define (call-next-method)
               ((if (Class-superclass ,c)
                    (vector-ref (Generic-dispatch-table ,g)
                                (Class-number (Class-superclass ,c)) )
                    (Generic-default ,g) )
                . ,(flat-variables variables) ) )
             . ,body ) )
         ',(cadr discriminant)
         ',(cdr call) ) ) ) ) )
```

The function `register-method` determines the class and the implicated generic function, converts the premethod into a method, verifies that the signatures are compatible, and installs it in the dispatch table. To test the compatibility of signatures during macro expansion would have required `define-method` to know the signatures of generic functions. Since this verification is simple and it is done only once at installation without hampering the efficiency of calls to generic functions, we have relegated it to evaluation. Notice that there is a comparison of functions by `eq?` for propagating methods.

```
(define (register-method generic-name pre-method class-name signature)
  (let* ((generic (->Generic generic-name))
         (class (->Class class-name))
         (new-method (pre-method generic class))
         (dispatch-table (Generic-dispatch-table generic))
         (old-method (vector-ref dispatch-table (Class-number class))) )
    (check-signature-compatibility generic signature)
    (let propagate ((cn (Class-number class)))
      (let ((content (vector-ref dispatch-table cn)))
        (if (eq? content old-method)
            (begin
              (vector-set! dispatch-table cn new-method)
              (for-each
               propagate
               (Class-subclass-numbers (number->class cn)) ) ) ) ) ) ) )
(define (check-signature-compatibility generic signature)
  (define (coherent-signatures? la lb)
    (if (pair? la)
        (if (pair? lb)
            (and (or ;;similar discriminating variable
```

```
                    (and (pair? (car la)) (pair? (car lb)))
                    ;;similar regular variable
                    (and (symbol? (car la)) (symbol? (car lb))) )
                (coherent-signatures? (cdr la) (cdr lb)) )
            #f )
        (or (and (null? la) (null? lb))
            ;;similar dotted variable
            (and (symbol? la) (symbol? lb)) ) ) )
    (if (not (coherent-signatures? (Generic-signature generic) signature))
      (meroonet-error "Incompatible signatures" generic signature) ) )
```

11.11 Conclusions

In the programs you've just seen, there are a lot of points that could be improved to increase the speed of MEROONET in Scheme or to heighten its reflective qualities. You can also imagine other improvements in a native implementation where every value would be a MEROONET object. For that reason, EULISP, ILOG TALK, and CLtL2 all integrate the concepts of objects in their basic foundations.

11.12 Exercises

Exercise 11.1 : Design a better, less approximate version of `Object?`.

Exercise 11.2 : Design a generic function, `clone`, to copy a MEROONET object. Make that a shallow copy.

Exercise 11.3 : We could create new types of classes, subclasses of `Class`, that we call metaclasses. Write a metaclass where the instances are classes that count the number of objects that they have created.

Exercise 11.4 : In order to heighten the reflective qualities of MEROONET, classes and fields could have supplementary fields to refer to the accompanying functions (predicate and allocators for classes; reader, writer, and length-getter for fields) that are associated with them. Extend MEROONET to do that.

Exercise 11.5 : CLOS does not demand that a generic function should already exist in order to add a new method to it. Modify `define-method` to create the generic function on the fly, if it doesn't exist already.

Exercise 11.6 : In some object systems like CLOS or EULISP, it is possible, conjointly with `call-next-method`, to know whether there is a method that follows, by means of `next-method?`. This reflective capacity makes it possible to reel off all the supermethods without error. The "method" that follows the one for `Object` corresponds to the default body, but `next-method?` replies False when there is no method other than that one. Modify `define-method` to get `next-method?`.

Recommended Reading

Another approach to objects in Scheme is inspired by that of T in [AR88]. For the fans of meta-objects, there is the historic article defining ObjVlisp, [Coi87] or the self-description of CLOS in [KdRB92].

Answers to Exercises

Exercise 1.1 : All you have to do is put trace commands in the right places, like this:

```
(define (tracing.evaluate exp env)
  (if ... ...
      (case (car exp) ...
        (else (let ((fn        (evaluate (car e) env))
                    (arguments (evlis (cdr e) env)) )x
                (display '(calling ,(car e) with . ,arguments)
                         *trace-port* )
                (let ((result (invoke fn arguments)))
                  (display '(returning from ,(car e) with ,result)
                           *trace-port* )
                  result ) )) ) ) )
```

Notice two points. First, the *name* of a function is printed, rather than its value. That convention is usually more informative. Second, print statements are sent out on a particular stream so that they can be more easily redirected to a window or log-file or even mixed in with the usual output stream.

Exercise 1.2 : In [Wan80b], that optimization is attributed to Friedman and Wise. The optimization lets us overlook the fact that there is still another `(evlis '() env)` to carry out when we evaluate the last term in a list. In practice, since the result is independent of `env`, there is no point in storing this call along with the value of `env`. Since lists of arguments are usually on the order of three or four terms, and since this optimization can always be carried out at the end of a list, it becomes a very useful one. Notice the local function that saves a test, too.

```
(define (evlis exps env)
  (define (evlis exps)
    ;;(assume (pair? exps))
    (if (pair? (cdr exps))
        (cons (evaluate (car exps) env)
              (evlis (cdr exps)) )
        (list (evaluate (car exps) env)) ) )
  (if (pair? exps)
      (evlis exps)
      '() ) )
```

Exercise 1.3 : This representation is known as the *rib cage* because of its obvious resemblance to that part of the body. It lowers the cost of a function call in the

number of pairs consumed, but it increases the cost for searching and for modifying the value of variables. Moreover, verifying the arity of the function is no longer so safe because we can no longer detect superfluous arguments this way. To do so, we have to modify **extend**.

```
(define (lookup id env)
  (if (pair? env)
      (let look ((names (caar env))
                 (values (cdar env)) )
        (cond ((symbol? names)
               (if (eq? names id) values
                   (lookup id (cdr env)) ) )
              ((null? names) (lookup id (cdr env)))
              ((eq? (car names) id)
               (if (pair? values)
                   (car values)
                   (wrong "Too less values") ) )
              (else (if (pair? values)
                        (look (cdr names) (cdr values))
                        (wrong "Too less values") )) ) ) )
      (wrong "No such binding" id) ) )
```

You can deduce the function **update!** straightforwardly from **lookup**.

Exercise 1.4 :

```
(define (s.make-function variables body env)
  (lambda (values current.env)
    (for-each (lambda (var val)
                (putprop var 'apval (cons val (getprop var 'apval))) )
              variables values )
    (let ((result (eprogn body current.env)))
      (for-each (lambda (var)
                  (putprop var 'apval (cdr (getprop var 'apval))) )
                variables )
      result ) ) )
(define (s.lookup id env)
  (car (getprop id 'apval)) )
(define (s.update! id env value)
  (set-car! (getprop id 'apval) value) )
```

Exercise 1.5 : Since there are many other predicates in the same predicament, just define a macro to define them. Notice that the value of **the-false-value** is True for the definition Lisp.

```
(define-syntax defpredicate
  (syntax-rules ()
    ((defpredicate name value arity)
     (defprimitive name
       (lambda values (or (apply value values) the-false-value))
       arity ) ) ) )
(defpredicate > > 2)
```

Exercise 1.6 : The hard part of this exercise is due to the fact that the arity of `list` is so variable. For that reason, you should define it directly by `definitial`, like this:

```
(definitial list
  (lambda (values) values) )
```

Exercise 1.7 : Well, of course, the obvious way to define `call/cc` is to use `call/cc`. The trick is to convert the underlying Lisp functions into functions in the Lisp being defined.

```
(defprimitive call/cc
  (lambda (f)
    (call/cc (lambda (g)
               (invoke
                f (list (lambda (values)
                          (if (= (length values) 1)
                              (g (car values))
                              (wrong "Incorrect arity" g) ) ))) )) )
  1 )
```

Exercise 1.8 : Here we have very much the same problem as in the definition of `call/cc`: you have to do the work between the abstractions of the underlying definition Lisp and the Lisp being defined. The function `apply` has variable arity, by the way, and it must transform its list of arguments (especially its last one) into an authentic list of values.

```
(definitial apply
  (lambda (values)
    (if (>= (length values) 2)
        (let ((f (car values))
              (args (let flat ((args (cdr values)))
                      (if (null? (cdr args))
                          (car args)
                          (cons (car args) (flat (cdr args))) ) )) )
          (invoke f args) )
          (wrong "Incorrect arity" 'apply) ) ) )
```

Exercise 1.9 : Just store the call continuation to the interpreter and bind this escape (conveniently adapted to the function calling protocol) to the variable `end`.

```
(define (chapter1d-scheme)
  (define (toplevel)
    (display (evaluate (read) env.global))
    (toplevel) )
  (display "Welcome to Scheme")(newline)
  (call/cc (lambda (end)
             (defprimitive end end 1)
             (toplevel) )) )
```

Exercise 1.10 : Of course, these comparisons depend simultaneously on at least two factors: the implementation that you are using and the programs that you are benchmarking. Normally, the comparisons show a ratio on the order of 5 to 15, according to [ITW86].
Even so, this exercise is useful to make you aware of the fact that the definition of `evaluate` is written in a fundamental Lisp, and it can be evaluated simulataneously by Scheme or by the language that `evaluate` defines.

Exercise 1.11 : Since you can always take sequences back to binary sequences, all you have to do here is to show how to rewrite them. The idea is to encapsulate expressions that risk inducing hooks inside *thunks*.

```
(begin expression1 expression2)
≡  ((lambda (void other) (other))
        expression1
        (lambda () expression2) )
```

Exercise 2.1 : It's straightforwardly translated as `(cons 1 2)`. For compatibility, you could also define this:

```
(define (funcall f . args) (apply f args))
(define (function f) f)
```

Or, again, you could define it with macros, like this:

```
(define-syntax funcall
  (syntax-rules ()
     ((funcall f arg ...) (f arg ...)) ) )
(define-syntax function
  (syntax-rules ()
     ((function f) f) ) )
```

Exercise 2.2 : The problem here is to know:

1. whether it is legal to talk about the function `bar` although it has not yet been defined;
2. in the case where `bar` has been defined before the result of `(function bar)` has been applied, whether you're going to get an error or the new function that has just been defined.

The difference is in the special form, `function`: are we talking about *the* function `bar` or about the value instantly associated with it in the name space for functions?

Exercise 2.3 : All you have to do is to modify `invoke` so that it recognizes numbers and lists in the function position. Do it like this:

```
(define (invoke fn args)
  (cond ((procedure? fn) (fn args))
        ((number? fn)
         (if (= (length args) 1)
             (if (>= fn 0) (list-ref (car args) fn)
                 (list-tail (car args) (- fn)) )
             (wrong "Incorrect arity" fn) ) )
```

```
        ((pair? fn)
         (map (lambda (f) (invoke f args))
              fn ) )
        (else (wrong "Cannot apply" fn)) ) )
```

Exercise 2.4 : The difficulty is that the function being passed belongs to the Lisp being defined, not to the definition Lisp.

```
(definitial new-assoc/de
  (lambda (values current.denv)
    (if (= 3 (length values))
        (let ((tag        (car values))
              (default    (cadr values))
              (comparator (caddr values)) )
          (let look ((denv current.denv))
            (if (pair? denv)
                (if (eq? the-false-value
                         (invoke comparator (list tag (caar denv))
                                            current.denv ) )
                    (look (cdr denv))
                    (cdar denv) )
                (invoke default (list tag) current.denv) ) ) )
        (wrong "Incorrect arity" 'assoc/de) ) ) )
```

Exercise 2.5 : Here again you have to adapt the function `specific-error` to the underlying exception system.

```
(define-syntax dynamic-let
  (syntax-rules ()
    ((dynamic-let () . body)
     (begin . body) )
    ((dynamic-let ((variable value) others ...) . body)
     (bind/de 'variable (list value)
              (lambda () (dynamic-let (others ...) . body)) ) ) ) )
(define-syntax dynamic
  (syntax-rules ()
    ((dynamic variable)
     (car (assoc/de 'variable specific-error)) ) ) )
(define-syntax dynamic-set!
  (syntax-rules ()
    ((dynamic-set! variable value)
     (set-car! (assoc/de 'variable specific-error) value) ) ) )
```

Exercise 2.6 : A private variable, `properties`, common to the two functions, contains the lists of properties for all the symbols.

```
(let ((properties '()))
  (set! putprop
        (lambda (symbol key value)
          (let ((plist (assq symbol properties)))
            (if (pair? plist)
```

```
                (let ((couple (assq key (cdr plist))))
                  (if (pair? couple)
                      (set-cdr! couple value)
                      (set-cdr! plist (cons (cons key value)
                                            (cdr plist) )) ) )
                (let ((plist (list symbol (cons key value))))
                  (set! properties (cons plist properties)) ) ) )
          value ) )
    (set! getprop
          (lambda (symbol key)
            (let ((plist (assq symbol properties)))
              (if (pair? plist)
                  (let ((couple (assq key (cdr plist))))
                    (if (pair? couple)
                        (cdr couple)
                        #f ) )
                  #f ) ) ) ) )
```

Exercise 2.7 : Just add the following clause to the interpreter, `evaluate`.

```
...
((label)  ;Syntax: (label name (lambda (variables) body))
 (let* ((name    (cadr e))
        (new-env (extend env (list name) (list 'void)))
        (def     (caddr e))
        (fun     (make-function (cadr def) (cddr def) new-env)) )
   (update! name new-env fun)
   fun ) )
```

Exercise 2.8 : Just add the following clause to `f.evaluate`. Notice the resemblance to `flet` except with respect to the function environment where local functions are created.

```
...
((labels)
 (let ((new-fenv (extend fenv
                         (map car (cadr e))
                         (map (lambda (def) 'void) (cadr e)) )))
   (for-each (lambda (def)
               (update! (car def)
                        new-fenv
                        (f.make-function (cadr def) (cddr def)
                                         env new-fenv ) ) )
             (cadr e) )
   (f.eprogn (cddr e) env new-fenv ) ) )
```

Exercise 2.9 : Since a `let` form preserves indetermination, you have to make sure not to sequence the computation of initialization forms and thus organize them into a `let`, which must however be in the right environment. You could do that using a hygienic expansion or a whole lot of **gensyms** for the temporary variables, $temp_i$.

```
(let ((variable1 'void)
      ...
      (variablen 'void) )
  (let ((temp1 expression1)
        ...
        (tempn expressionn) )
    (set! variable1 temp1)
    ...
    (set! variablen tempn)
    corps ) )
```

Exercise 2.10 : Here's a binary version. The η-conversion has been modified to take a binary function into account.

```
(define fix2
  (let ((d (lambda (w)
             (lambda (f)
               (f (lambda (x y) (((w w) f) x y))) ) )))
    (d d) ) )
```

Last but not least, here is an n-ary version.

```
(define fixN
  (let ((d (lambda (w)
             (lambda (f)
               (f (lambda args (apply ((w w) f) args))) ) )))
    (d d) ) )
```

Exercise 2.11 : Perhaps a little intuition is enough to see that the preceding definition for `fixN` can be extended, like this:

```
(define NfixN
  (let ((d (lambda (w)
             (lambda (f*)
               (list ((car f*)
                      (lambda a (apply (car ((w w) f*)) a))
                      (lambda a (apply (cadr ((w w) f*)) a)) )
                     ((cadr f*)
                      (lambda a (apply (car ((w w) f*)) a))
                      (lambda a (apply (cadr ((w w) f*)) a)) ) ) ) )))
    (d d) ) )
```

Now be careful to prevent `((w w) f)` being evaluated too hastily, and then generalize the extension in this way:

```
(define NfixN2
  (let ((d (lambda (w)
             (lambda (f*)
               (map (lambda (f)
                      (apply f (map (lambda (i)
                                      (lambda a (apply
                                                  (list-ref ((w w) f*) i)
                                                  a )) )
                                    (iota 0 (length f*)) )) )
```

```
                    f* ) ) )))
    (d d) ) )
```

Careful: the order of the functions for which you take the fixed point must always be the same. Consequently, the definition of `odd?` must be first in the list and the functional that defines it must have `odd?` as its first variable.
The function `iota` is thus defined in a way that recalls APL, like this:

```
(define (iota start end)
  (if (< start end)
      (cons start (iota (+ 1 start) end))
      '() ) )
```

Exercise 2.12 : That function is attributed to Klop in [Bar84]. You can verify that `((klop meta-fact) 5)` really does compute 120.
Since all the internal variables `s`, `c`, `h`, `e`, and `m` are bound to `r`, their order does not matter much in the application `(m e c h e s)`. It is sufficient for the arity to be respected. You can even add or cut back on variables. If you reduce `s`, `c`, `h`, `e`, and `m` to nothing more than `w`, then you get **Y**.

Exercise 2.13 : The answer is 120. Isn't it nice that auto-application makes recursion possible here? Another possibility would have been to modify the code for the factorial to that it takes itself as an argument. That would lead to this:

```
(define (factfact n)
  (define (internal-fact f n)
    (if (= n 1) 1
        (* n (f f (- n 1))) ) )
  (internal-fact internal-fact n) )
```

Exercise 3.1 : That form could be named `(the-current-continuation)` because it returns its own continuation. There we see the point of `call/cc` with respect to `the-continuation` since `the-continuation` brings back only its continuation, a poor thing of no interest to us. Let's look at the details of a computation, and as we do so, let's index the continuations and functions that come into play; we'll abbreviate the abbreviation `call/cc` as just `cc` here. Continuations will prefix expressions as an index, so we'll calculate ${}_{k_0}$`(call/cc`$_1$ `call/cc`$_2$`)`; k_0 is the continuation with which to calculate `(cc`$_1$ `cc`$_2$`)`. Remember that the definition of `call/cc` is this:

$${}_k\texttt{(call/cc }\phi\texttt{)} \quad \rightarrow \quad {}_k(\phi\ k)$$

Thus we have that ${}_{k_0}$`(cc`$_1$ `cc`$_2$`)` is rewritten as ${}_{k_0}$`(cc`$_2$ `k`$_0$`)`, which in turn is rewritten as ${}_{k_0}$`(k`$_0$ `k`$_0$`)`, which returns the value k_0.

Exercise 3.2 : We'll use the same conventions as the preceding exercise and index this expression like this: ${}_{k_0}$`((cc`$_1$ `cc`$_2$`) (cc`$_3$ `cc`$_4$`))`. For simplicity, let's assume that the terms of a functional application are evaluated from left to right. Then the original expression becomes this: ${}_{k_0}$`(`$_{k_1}$`(cc`$_1$ `cc`$_2$`) (cc`$_3$ `cc`$_4$`))` where k_1 is $\lambda\phi.{}_{k_0}$`(`ϕ ${}_{k_2}$`(cc`$_3$ `cc`$_4$`))` and k_2 is $\lambda\epsilon.{}_{k_0}$`(`k_1 ϵ`)`. The initial form is rewritten as ${}_{k_0}(k_1\ k_2)$, that is, ${}_{k_0}$`(`k_2 ${}_{k'_2}$`(cc`$_3$ `cc`$_4$`))` where k'_2 is $\lambda\epsilon.{}_{k_0}(k_2\ \epsilon)$. That leads to ${}_{k_0}(k_1\ k'_2)$

and then to $_{k_0}(k'_1\ k''_2)$ ... So you see that the calculation loops indefinitely. You can verify that this looping does not depend on the order of evaluation of the terms of the functional application. This may very well be the shortest possible program, measured in number of terms, that will loop indefinitely.

Exercise 3.3 : Each computation prefixed by a *label* will be transformed into a local function within a huge form, `labels`. The `go`s are translated into calls to those functions, but they are associated with a dynamic escape to insure that the `go` forms have the right continuation. The expansion looks like this:

```
(block EXIT
  (let (LABEL (TAG (list 'tagbody)))
    (labels ((INIT () expressions0... (label1))
             (label1 () expressions1... (label2))
             ...
             (labeln () expressionsn... (return-from EXIT nil)) )
      (setq LABEL (function INIT))
      (while #t
        (setq LABEL (catch TAG (funcall LABEL))) ) ) ) )
```

The forms (`go` *label*) are translated into (`throw TAG` *label*); (`return` *value*) will become (`return-from EXIT` *value*). In the preceding lines, those names are written in all capital letters to avoid conflicts.

A fairly complicated translation of `go` insures the right continuation of the branching, and in `(bar (go L))`, it prevents `bar` from being called when `(go L)` returns a value. That unfortunately would have happened if we had carelessly written this, for example:

```
(tagbody   A (return (+ 1 (catch 'foo (go B))))
           B (* 2 (throw 'foo 5)) )
```

See also [Bak92c].

Exercise 3.4 : Introduce a new subclass of functions:

```
(define-class function-with-arity function (arity))
```

Then redefine the evaluation of `lambda` forms so that they now create one such instance:

```
(define (evaluate-lambda n* e* r k)
  (resume k (make-function-with-arity n* e* r (length n*))) )
```

Now adapt the invocation protocol for these new functions, like this:

```
(define-method (invoke (f function-with-arity) v* r k)
  (if (= (function-with-arity-arity f) (length v*))
      (let ((env (extend-env (function-env f)
                             (function-variables f)
                             v* )))
        (evaluate-begin (function-body f) env k) )
      (wrong "Incorrect arity" (function-variables f) v*) ) )
```

Exercise 3.5 :

```
(definitial apply
  (make-primitive
   'apply
   (lambda (v* r k)
     (if (>= (length v*) 2)
         (let ((f (car v*))
               (args (let flat ((args (cdr v*)))
                       (if (null? (cdr args))
                           (car args)
                           (cons (car args) (flat (cdr args))) ) )) )
           (invoke f args r k) )
         (wrong "Incorrect arity" 'apply) ) ) ) )
```

Exercise 3.6 : Take a little inspiration from the preceding exercise and define a new type of function, like this:

```
(define-class function-nadic function (arity))
(define (evaluate-lambda n* e* r k)
  (resume k (make-function-with-arity n* e* r (length n*))) )
(define-method (invoke (f function-nadic) v* r k)
  (define (extend-env env names values)
    (if (pair? names)
        (make-variable-env
         (extend-env env (cdr names) (cdr values))
         (car names)
         (car values) )
        (make-variable-env env names values) ) )
  (if (>= (length v*) (function-nadic-arity f))
      (let ((env (extend-env (function-env f)
                             (function-variables f)
                             v* )))
        (evaluate-begin (function-body f) env k) )
      (wrong "Incorrect arity" (function-variables f) v*) ) )
```

Exercise 3.7 : Put the interaction loop in the initial continuation. For example, you could write this:

```
(define (chap3j-interpreter)
  (letrec ((k.init (make-bottom-cont
                    'void (lambda (v) (display v)
                                      (toplevel) ) ))
           (toplevel (lambda () (evaluate (read) r.init k.init))) )
    (toplevel) ) )
```

Exercise 3.8 : Define the class of reified continuations; all it does is encapsulate three things: an internal continuation (an instance of `continuation`), the function `call/cc` to build such an object, and the right method of invocation.

```
(define-class reified-continuation value (k))
(definitial call/cc
```

```
(make-primitive 'call/cc
                (lambda (v* r k)
                  (if (= 1 (length v*))
                      (invoke (car v*)
                              (list (make-reified-continuation k))
                              r
                              k )
                      (wrong "Incorrect arity" 'call/cc v*) ) ) ) )
(define-method (invoke (f reified-continuation) v* r k)
  (if (= 1 (length v*))
      (resume (reified-continuation-k f) (car v*))
      (wrong "Continuations expect one argument" v* r k) ) )
```

Exercise 3.9 : This point of this function is never to return.

COMMON LISP

```
(defun eternal-return (thunk)
  (labels ((loop ()
             (unwind-protect (thunk)
               (loop) ) ))
    (loop) ) )
```

Exercise 3.10 : The values of those expressions are 33 and 44. The function **make-box** simulates a box with no apparent assignment nor function side effects. We get this effect by combining **call/cc** and **letrec**. If you think of the expansion of **letrec** in terms of **let** and **set!**, then it is easier to see how we get that. However, it makes life much more difficult for partisans of the special form **letrec** in the presence of first class continuations that can be invoked more than once.

Exercise 3.11 : First, rewrite **evaluate** simply as a generic function, like this:

```
(define-generic (evaluate (e) r k)
   (wrong "Not a program" e r k) )
```

Then take the existing functions to ornament **evaluate**, like this:

```
(define-method (evaluate (e quotation) r k)
  (evaluate-quote (quotation-value e) r k) )
(define-method (evaluate (e assignment) r k)
  (evaluate-set! (assignment-name e)
                 (assignment-form e)
                 r k ) )
...
```

Then you still have to define classes corresponding to the various possible syntactic forms, like this:

```
(define-class program Object ())
(define-class quotation program (value))
(define-class assignment program (name form))
...
```

Now the only thing left to do is to define an appropriate reader that knows how to read a program and build an instance of **program**. That's the purpose of the function **objectify** that you'll see in Section 9.11.1.

Exercise 3.12 : To define `throw` as a function, do this:

```
(definitial throw
   (make-primitive
    'throw
    (lambda (v* r k)
      (if (= 2 (length v*))
          (catch-lookup k (car v*)
                        (make-throw-cont k `(quote ,(cadr v*)) r) )
          (wrong "Incorrect arity" 'throw v*) ) ) ) )
```

So that we didn't simply define a variation on `catch-lookup`, we fabricated a false instance of `throw-cont` to trick the interpreter. The only important value there is the one to transmit.

Exercise 3.13 : Code translated into CPS is slower because it creates many closures to simulate continuation. By the way, a transformation into CPS is not idempotent. That is, translating code into CPS and then translating that result into CPS does not yield the identity. That's obvious when we consider the factorial in CPS. The continuation `k` can be any function at all and may also have side effects on control. For example, we could write this:

```
(define (cps-fact n k)
  (if (= n 0) (k 1) (cps-fact (- n 1) (lambda (v) (k (* n v))))) )
```

The function `cps-fact` can be invoked with a rather special continuation, as in this example:

```
(call/cc (lambda (k) (* 2 (cps-fact 4 k)))) → 24
```

Exercise 3.14 : The function `the-current-continuation` could also be defined as in Exercise 3.1, like this:

```
(define (cc f)
  (let ((reified? #f))
    (let ((k (the-current-continuation)))
      (if reified? k (begin (set! reified? #t) (f k))) ) ) )
```

Special thanks to Luc Moreau, who suggested this exercise in [Mor94].

Exercise 4.1 : Many techniques exist; among them, return partial results or use continuations. Here are two solutions:

```
(define (min-max1 tree)
  (define (mm tree)
    (if (pair? tree)
        (let ((a (mm (car tree)))
              (d (mm (cdr tree))) )
          (list (min (car a) (car d))
                (max (cadr a) (cadr d)) ) )
        (list tree tree) ) )
  (mm tree) )
(define (min-max2 tree)
  (define (mm tree k)
    (if (pair? tree)
```

```
        (mm (car tree)
            (lambda (mina maxa)
              (mm (cdr tree)
                  (lambda (mind maxd)
                    (k (min mina mind)
                       (max maxa maxd) ) ) ) ) )
        (k tree tree) ) )
  (mm tree list) )
```

That first solution consumes a lot of lists that are promptly forgotten. For this kind of algorithm, you can imagine carrying out transformations that eliminate this kind of abusive consummation; those transformations are known in [Wad88] as *deforestation* The second solution consumes a lot of closures. The version in this book is much more efficient, in spite of its side effects (though the side effects might make it intolerable to certain people).

Exercise 4.2 : The following functions are prefixed by q to distinguish them from the corresponding primitive functions.

```
(define (qons a d) (lambda (msg) (msg a d)))
(define (qar pair) (pair (lambda (a d) a)))
(define (qdr pair) (pair (lambda (a d) d)))
```

Exercise 4.3 : Use the idea that two dotted pairs are the same if a modification of one is visible in the other.

```
(define (pair-eq? a b)
  (let ((tag (list 'tag))
        (original-car (car a)) )
    (set-car! a tag)
    (let ((result (eq? (car b) tag)))
      (set-car! a original-car)
      result ) ) )
```

Exercise 4.4 : Assuming you've already added a clause to **evaluate** to recognize the special form **or**,

```
...
((or) (evaluate-or (cadr e) (caddr e) r s k)) ...
```

then do something like this:

```
(define (evaluate-or e1 e2 r s k)
  (evaluate e1 r s (lambda (v ss)
                     (((v 'boolify)
                       (lambda () (k v ss))
                       (lambda () (evaluate e2 r s k)) ) ) )) )
```

Notice that β is evaluated with the memory **s** not **ss**.

Exercise 4.5 : Actually that exercise is a little ambiguous. Here are two solutions; one returns the value that the variable had before the calculation of the value to assign; the other, after that calculation.

```
(define (new1-evaluate-set! n e r s k)
  (evaluate e r s
    (lambda (v ss)
      (k (ss (r n)) (update ss (r n) v)) ) ) )
(define (new2-evaluate-set! n e r s k)
  (evaluate e r s
    (lambda (v ss)
      (k (s (r n)) (update ss (r n) v)) ) ) )
```

Those two programs give different results for the following expression:

```
(let ((x 1))
  (set! x (set! x 2)) )
```

Exercise 4.6 : The only problem with **apply** is that the last argument is a list of the Scheme interpreter that we have to decode into values. `-11` is a label to help us identify the primitive **apply**.

```
(definitial apply
  (create-function
   -11 (lambda (v* s k)
         (define (first-pairs v*)
           ;;(assume (pair? v*))
           (if (pair? (cdr v*))
               (cons (car v*) (first-pairs (cdr v*)))
               '() ) )
         (define (terms-of v s)
           (if (eq? (v 'type) 'pair)
               (cons (s (v 'car)) (terms-of (s (v 'cdr)) s))
               '() ) )
         (if (>= (length v*) 2)
             (if (eq? ((car v*) 'type) 'function)
                 (((car v*) 'behavior)
                  (append (first-pairs (cdr v*))
                          (terms-of (car (last-pair (cdr v*))) s) )
                  s k )
                 (wrong "First argument not a function") )
             (wrong "Incorrect arity for apply") ) ) ) )
```

For `call/cc`, we allocate a label for each continuation to make it unique.

```
(definitial call/cc
  (create-function
   -13 (lambda (v* s k)
         (if (= 1 (length v*))
             (if (eq? ((car v*) 'type) 'function)
                 (allocate 1 s
                  (lambda (a* ss)
                    (((car v*) 'behavior)
                     (list (create-function
                            (car a*)
                            (lambda (vv* sss kk)
                              (if (= 1 (length vv*))
                                  (k (car vv*) sss)
```

```
                                        (wrong "Incorrect arity") ) ) ) )
                                ss k ) ) )
                            (wrong "Non functional argument for call/cc") )
                        (wrong "Incorrect arity for call/cc") ) ) ) )
```

Exercise 4.7 : The difficulty is to test the compatibility of the arity and to change a list (of the Scheme interpreter) into a list of values of the Scheme being interpreted.

```
(define (evaluate-nlambda n* e* r s k)
  (define (arity n*)
    (cond ((pair? n*) (+ 1 (arity (cdr n*))))
          ((null? n*) 0)
          (else       1) ) )
  (define (update-environment r n* a*)
    (cond ((pair? n*) (update-environment
                        (update r (car n*) (car a*)) (cdr n*) (cdr a*) ))
          ((null? n*) r)
          (else (update r n* (car a*))) ) )
  (define (update-store s a* v* n*)
    (cond ((pair? n*) (update-store (update s (car a*) (car v*))
                                    (cdr a*) (cdr v*) (cdr n*) ))
          ((null? n*) s)
          (else (allocate-list v* s (lambda (v ss)
                                      (update ss (car a*) v) ))) ) )
  (allocate 1 s
    (lambda (a* ss)
      (k (create-function
          (car a*)
          (lambda (v* s k)
            (if (compatible-arity? n* v*)
                (allocate (arity n*) s
                 (lambda (a* ss)
                   (evaluate-begin e*
                                   (update-environment r n* a*)
                                   (update-store ss a* v* n*)
                                   k ) ) )
                (wrong "Incorrect arity") ) ) )
         ss ) ) ) )
(define (compatible-arity? n* v*)
  (cond ((pair? n*) (and (pair? v*)
                         (compatible-arity? (cdr n*) (cdr v*)) ))
        ((null? n*) (null? v*))
        ((symbol? n*) #t) ) )
```

Exercise 5.1 : Prove it by induction on the number of terms in the application.

Exercise 5.2 :

$\mathcal{L}[\![(\texttt{label}\ \nu\ \pi)]\!]\rho = (\mathbf{Y}\ \lambda\varepsilon.(\mathcal{L}[\![\pi]\!]\ \rho[\nu \rightarrow \varepsilon]))$

Exercise 5.3 :

$\mathcal{E}[\![$(dynamic ν)$]\!]\rho\delta\kappa\sigma =$
let $\varepsilon = (\delta\ \nu)$
in **if** $\varepsilon =$ *no-such-dynamic-variable*
then **let** $\alpha = (\gamma\ \nu)$
in **if** $\alpha =$ *no-such-global-variable*
then *wrong* "No such variable"
else $(\kappa\ (\sigma\ \alpha)\ \sigma)$
endif
else $(\kappa\ \varepsilon\ \sigma)$
endif

Exercise 5.4 : The macro will transform the application into a series of thunks that we evaluate in a non-prescribed order implemented by the function `determine!`.

```
(define-syntax unordered
  (syntax-rules ()
    ((unordered f) (f))
    ((unordered f arg ...)
     (determine! (lambda () f) (lambda () arg) ... ) ) ) )
(define (determine! . thunks)
  (let ((results (iota 0 (length thunks))))
    (let loop ((permut (random-permutation (length thunks))))
      (if (pair? permut)
          (begin (set-car! (list-tail results (car permut))
                           (force (list-ref thunks (car permut))) )
                 (loop (cdr permut)) )
          (apply (car results) (cdr results)) ) ) ) )
```

Notice that the choice of the permutation is made at the beginning, so this solution is not really as good as the denotation used in this chapter. If the function `random-permutation` is defined like this:

```
(define (random-permutation n)
  (shuffle (iota 0 n)) )
```

then we could make the choice of the permutation dynamic by means of `d.determine!`.

```
(define (d.determine! . thunks)
  (let ((results (iota 0 (length thunks))))
    (let loop ((permut (shuffle (iota 0 (length thunks)))))
      (if (pair? permut)
          (begin (set-car! (list-tail results (car permut))
                           (force (list-ref thunks (car permut))) )
                 (loop (shuffle (cdr permut))) )
          (apply (car results) (cdr results)) ) ) ) )
```

Exercise 6.1 : A simple way of doing that is to add a supplementary argument to the combinator to indicate which variable (either a symbol or simply its name), like this:

```
(define (CHECKED-GLOBAL-REF- i n)
  (lambda ()
    (let ((v (global-fetch i)))
      (if (eq? v undefined-value)
          (wrong "Uninitialized variable" n)
          v ) ) ) )
```

However, that solution increases the overall size of the code generated. A better solution is to generate a symbol table so that the name of the faulty variable would be there.

```
(define sg.current.names (list 'foo))
(define (stand-alone-producer e)
  (set! g.current (original.g.current))
  (let* ((m (meaning e r.init #t))
         (size (length g.current))
         (global-names (map car (reverse g.current))) )
    (lambda ()
      (set! sg.current (make-vector size undefined-value))
      (set! sg.current.names global-names)
      (set! *env* sr.init)
      (m) ) ) )
(define (CHECKED-GLOBAL-REF+ i)
  (lambda ()
    (let ((v (global-fetch i)))
      (if (eq? v undefined-value)
          (wrong "Uninitialized variable" (list-ref sg.current.names i))
          v ) ) ) )
```

Exercise 6.2 : The function `list` is, of course, just `(lambda l l)`, so all you have to do is use this definition and play around with the combinators, like this:

```
(definitial list ((NARY-CLOSURE (SHALLOW-ARGUMENT-REF 0) 0)))
```

Exercise 6.3 : First, you can redefine every combinator c as `(lambda args '(`c `. ,args))`. Then all you have to do is print the results of pretreatment.

Exercise 6.4 : The most direct way to get there is to modify the evaluation order for arguments of applications and thus adopt right to left order.

```
(define (FROM-RIGHT-STORE-ARGUMENT m m* rank)
  (lambda ()
    (let* ((v* (m*))
           (v (m)) )
      (set-activation-frame-argument! v* rank v)
      v* ) ) )
(define (FROM-RIGHT-CONS-ARGUMENT m m* arity)
  (lambda ()
    (let* ((v* (m*))
           (v (m)) )
      (set-activation-frame-argument!
       v* arity (cons v (activation-frame-argument v* arity)) )
```

```
        v* ) ) )
```

You could also modify **meaning*** to preserve the evaluation order but to allocate the record before computing the arguments. By the way, computing the function term before computing the arguments (regardless of the order in which they are evaluated) lets you look at the closure you get, for example, to allocate an activation record adapted to the number of temporary variables required. Then there will be only one allocation, not two.

Exercise 6.5 : Insert the following line as a special form in **meaning**:

```
... ((redefine) (meaning-redefine (cadr e))) ...
```

Then redefine its effect, like this:

```
(define (meaning-redefine n)
  (let ((kind1 (global-variable? g.init n)))
    (if kind1
        (let ((value (vector-ref sg.init (cdr kind1))))
          (let ((kind2 (global-variable? g.current n)))
            (if kind2
                (static-wrong "Already redefined variable" n)
                (let ((index (g.current-extend! n)))
                  (vector-set! sg.current index value) ) ) ) )
        (static-wrong "No such variable to redefine" n) )
    (lambda () 2001) ) )
```

That redefinition occurs during pretreatment, not during execution. The form **redefine** returns just any value.

Exercise 6.6 : A function without variables does not need to extend the current environment. Extending the current environment slowed down access to deep variables. Change **meaning-fix-abstraction** so that it detects thunks, and define a new combinator to do that.

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let ((arity (length n*)))
    (if (= arity 0)
        (let ((m+ (meaning-sequence e+ r #t)))
          (THUNK-CLOSURE m+) )
        (let* ((r2 (r-extend* r n*))
               (m+ (meaning-sequence e+ r2 #t)) )
          (FIX-CLOSURE m+ arity) ) ) ) )
(define (THUNK-CLOSURE m+)
  (let ((arity+1 (+ 0 1)))
    (lambda ()
      (define (the-function v* sr)
        (if (= (activation-frame-argument-length v*) arity+1)
            (begin (set! *env* sr)
                   (m+) )
            (wrong "Incorrect arity") ) )
      (make-closure the-function *env*) ) ) )
```

Exercise 7.1 : First, create the register, like this:

```
(define *dynenv* -1)
```

Then modify the functions that preserve the environment, like this:

```
(define (preserve-environment)
  (stack-push *dynenv*)
  (stack-push *env*) )
(define (restore-environment)
  (set! *env* (stack-pop))
  (set! *dynenv* (stack-pop)) )
```

There's hardly anything left to do as far as finding the dynamic environment now, but the way to handle the stack has changed.

```
(define (search-dynenv-index)
  *dynenv* )
(define (pop-dynamic-binding)
  (stack-pop)
  (stack-pop)
  (set! *dynenv* (stack-pop)) )
(define (push-dynamic-binding index value)
  (stack-push *dynenv*)
  (stack-push value)
  (stack-push index)
  (set! *dynenv* (- *stack-index* 1)) )
```

Exercise 7.2 : The function is simple:

```
(definitial load
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let ((filename (activation-frame-argument *val* 0)))
             (set! *pc* (install-object-file! filename)) )
           (signal-exception
            #t (list "Incorrect arity" 'load) ) ) ) ) ) )
```

However, that definition poses a few problems. Analyze, for example, how a captured continuation returns and restarts when a file is being loaded. For example,

```
(display 'attention)
(call/cc (lambda (k) (set! *k* k)))
(display 'caution)
```

After this file has been loaded, will invoking the continuation `*k*` reprint the symbol `caution`? Yes, with this implementation!

Also notice that if we load the compiled file defining a global variable, say, `bar`, which was unknown to the application, it will remain unknown to the application.

Exercise 7.3 : Here's the function:

```
(definitial global-value
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (define (get-index name)
      (let ((where (memq name sg.current.names)))
        (if where (- (length where) 1)
            (signal-exception
             #f (list "Undefined global variable" name) ) ) ) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let* ((name (activation-frame-argument *val* 0))
                  (i (get-index name)) )
             (set! *val* (global-fetch i))
             (when (eq? *val* undefined-value)
               (signal-exception #f (list "Uninitialized variable" i)) )
             (set! *pc* (stack-pop)) )
           (signal-exception
            #t (list "Incorrect arity" 'global-value) ) ) ) ) ) )
```

Since we have re-introduced the possibility of an non-existing variable, of course, we have to test whether the variable has been initialized.

Exercise 7.4 : First, add the allocation of a vector to the function **run-machine** to store the current value of dynamic variables.

```
... (set! *dynamics* (make-vector (+ 1 (length dynamics))
                                  undefined-value )) ;NEW
```

Then redefine the appropriate access functions, like this:

```
(define (find-dynamic-value index)
  (let ((v (vector-ref *dynamics* index)))
    (if (eq? v undefined-value)
        (signal-exception #f (list "No such dynamic binding" index))
        v ) ) )
(define (push-dynamic-binding index value)
  (stack-push (vector-ref *dynamics* index))
  (stack-push index)
  (vector-set! *dynamics* index value) )
(define (pop-dynamic-binding)
  (let* ((index (stack-pop))
         (old-value (stack-pop)) )
    (vector-set! *dynamics* index old-value) ) )
```

Alas! that implementation is unfortunately incorrect because immediate access takes advantage of the fact that saved values are now found on the stack. A continuation capture gets only the values to restore, not the current values. When an escape suppresses a slice of the stack, it fails to restore the dynamic variables to what they were when the form **bind-exit** began to be evaluated. To correct all that, we must have a form like **unwind-protect** or more simply we must abandon

superficial implementation for a deeper implementation that doesn't pose that kind of problem and can be extended naturally in parallel.

Exercise 7.5 : With the following renamer, you can even interchange two variables by writing a list of substitutions such as `((fact fib) (fib fact))`, but look out: this is a dangerous game to play!

```
(define (build-application-renaming-variables
         new-application-name application-name substitutions )
  (if (probe-file application-name)
      (call-with-input-file application-name
        (lambda (in)
          (let* ((dynamics     (read in))
                 (global-names (read in))
                 (constants    (read in))
                 (code         (read in))
                 (entries      (read in)) )
            (close-input-port in)
            (write-result-file
             new-application-name
             (list ";;;renamed variables from " application-name)
             dynamics
             (let sublis ((global-names global-names))
               (if (pair? global-names)
                   (cons (let ((s (assq (car global-names)
                                        substitutions )))
                           (if (pair? s) (cadr s)
                               (car global-names) ) )
                         (sublis (cdr global-names)) )
                   global-names ) )
             constants
             code
             entries ) ) ) )
      (signal #f (list "No such file" application-name)) ) )
```

Exercise 7.6 : Be careful to modify the right instruction!

```
(define-instruction (CHECKED-GLOBAL-REF i) 8
  (set! *val* (global-fetch i))
  (if (eq? *val* undefined-value)
    (signal-exception #t (list "Uninitialized variable" i))
    (vector-set! *code* (- *pc* 2) 7) ) )
```

Exercise 8.1 : The test is not necessary because it doesn't recognize two variables by the same name. That convention prevents a list of variables from being cyclic.

Exercise 8.2 : Here's the hint:

```
(define (prepare e)
  (eval/ce `(lambda () ,e)) )
```

Exercise 8.3 :

```
(define (eval/at e)
  (let ((g (gensym)))
    (eval/ce `(lambda (,g) (eval/ce ,g))) ) )
```

Exercise 8.4 : Yes, by programming an appropriate error handler, like this:

```
(set! variable-defined?
      (lambda (env name)
        (bind-exit (return)
          (monitor (lambda (c ex) (return #f))
            (eval/b name env)
            #t ) ) ) )
```

Exercise 8.5 : We will quietly skip over how to handle the special form **monitor**, which appears in the definition of the reflective interpreter. After all, if we don't make any mistakes, **monitor** behaves just like **begin**. What follows here does not exactly conform to Scheme because the definition of special forms requires their names to be used as variables (not exactly legal, we know). However, that works in many implementations of Scheme. For the form **the-environment**, we will define it to capture the necessary bindings.

```
(define-syntax the-environment
  (syntax-rules ()
    ((the-environment)
     (capture-the-environment make-toplevel make-flambda flambda?
      flambda-behavior prompt-in prompt-out exit it extend error
      global-env toplevel eval evlis eprogn reference quote if set!
      lambda flambda monitor ) ) ) )
(define-syntax capture-the-environment
  (syntax-rules ()
    ((capture-the-environment word ...)
     (lambda (name . value)
       (case name
         ((word) ((handle-location word) value)) ...
         ((display) (if (pair? value)
                        (wrong "Immutable" 'display)
                        show ))
         (else (if (pair? value)
                   (set-top-level-value! name (car value))
                   (top-level-value name) )) ) ) ) ) )
(define-syntax handle-location
  (syntax-rules ()
    ((handle-location name)
     (lambda (value)
       (if (pair? value)
           (set! name (car value))
           name ) ) ) ) )
```

Finally we'll define the functions to handle the first class environment; they are **variable-defined?**, **variable-value**, and **set-variable-value!**.

```
(define undefined (cons 'un 'defined))
(define-class Envir Object
  ( name value next ) )
(define (enrich env . names)
  (let enrich ((env env)(names names))
    (if (pair? names)
        (enrich (make-Envir (car names) undefined env) (cdr names))
        env ) ) )
(define (variable-defined? name env)
  (if (Envir? env)
      (or (eq? name (Envir-name env))
          (variable-defined? name (Envir-next env)) )
      #t ) )
(define (variable-value name env)
  (if (Envir? env)
      (if (eq? name (Envir-name env))
          (let ((value (Envir-value env)))
            (if (eq? value undefined)
                (error "Uninitialized variable" name)
                value ) )
          (variable-value name (Envir-next env)) )
      (env name) ) )
```

As you see, the environment is a linked list of nodes ending with a closure. Now the reflective interpreter can run!

Exercise 9.1 : Use the hygienic macros of Scheme to write this:

```
(define-syntax repeat1
  (syntax-rules (:while :unless :do)
    ((_ :while p :unless q :do body ...)
     (let loop ()
       (if p (begin (if (not q) (begin body ...))
                    (loop) )) ) ) ) )
```

You could also use **define-abbreviation** directly to write this instead:

```
(with-aliases ((+let let) (+begin begin) (+when when) (+not not))
  (define-abbreviation (repeat2 . parms)
    (let ((p    (list-ref parms 1))
          (q    (list-ref parms 3))
          (body (list-tail parms 5))
          (loop (gensym)) )
      `(,+let ,loop ()
            (,+when ,p (,+begin (,+when (,+not ,q) . ,body)
                                (,loop) )) ) ) ) )
```

Exercise 9.2 : The difficulty here is to do arithmetic with the macro language of Scheme. One approach is to generate calls at execution to the function **length** on lists of selected lengths.

```
(define-syntax enumerate
  (syntax-rules ()
```

```
    ((enumerate) (display 0))
    ((enumerate e1 e2 ...)
     (begin (display 0) (enumerate-aux e1 (e1) e2 ...) ) ) ) )
(define-syntax enumerate-aux
  (syntax-rules ()
    ((enumerate-aux e1 len) (begin e1 (display (length 'len))))
    ((enumerate-aux e1 len e2 e3 ...)
     (begin e1 (display (length 'len))
            (enumerate-aux e2 (e2 . len) e3 ...) ) ) ) )
```

Exercise 9.3 : All you have to do is modify the function `make-macro-environment` so that all the levels are melded into one, like this:

```
(define (make-macro-environment current-level)
  (let ((metalevel (delay current-level)))
    (list (make-Magic-Keyword 'eval-in-abbreviation-world
           (special-eval-in-abbreviation-world metalevel) )
          (make-Magic-Keyword 'define-abbreviation
           (special-define-abbreviation metalevel))
          (make-Magic-Keyword 'let-abbreviation
           (special-let-abbreviation metalevel))
          (make-Magic-Keyword 'with-aliases
           (special-with-aliases metalevel) ) ) ) )
```

Exercise 9.4 : Writing the converter is a cake-walk. The only real point of interest is how to rename the variables. We'll keep an A-list to store the correspondances.

```
(define-generic (->Scheme (e) r))
(define-method (->Scheme (e Alternative) r)
  `(if ,(->Scheme (Alternative-condition e) r)
       ,(->Scheme (Alternative-consequent e) r)
       ,(->Scheme (Alternative-alternant e) r) ) )
(define-method (->Scheme (e Local-Assignment) r)
  `(set! ,(->Scheme (Local-Assignment-reference e) r)
         ,(->Scheme (Local-Assignment-form e) r) ) )
(define-method (->Scheme (e Reference) r)
  (variable->Scheme (Reference-variable e) r) )
(define-method (->Scheme (e Function) r)
  (define (renamings-extend r variables names)
    (if (pair? names)
        (renamings-extend (cons (cons (car variables) (car names)) r)
                          (cdr variables) (cdr names) )
        r ) )
  (define (pack variables names)
    (if (pair? variables)
        (if (Local-Variable-dotted? (car variables))
            (car names)
            (cons (car names) (pack (cdr variables) (cdr names))) )
        '() ) )
  (let* ((variables (Function-variables e))
```

```
        (new-names (map (lambda (v) (gensym))
                        variables ))
        (newr (renamings-extend r variables new-names)) )
    `(lambda ,(pack variables new-names)
       ,(->Scheme (Function-body e) newr)) ) )
(define-generic (variable->Scheme (e) r))
```

Exercise 9.5 : As it's written, MEROONET mixes the two worlds of expansion and execution; several resources belong to both worlds simultaneously. For example, `register-class` is invoked at expansion and when a file is loaded.

Exercise 10.1 : All you have to change is the function `SCM_invoke` and the calling protocol for closures of minor fixed arity. It suffices to take the one for primitives of fixed arity—just don't forget to provide the object representing the closure as the first argument. You must also change the interface for generated C functions to make them adopt the signature you've just adopted.

Exercise 10.2 : Refine global variables so that they have a supplementary field indicating whether they have really been intialized or not. In consequence of that design change, you'll have to change how free global variables are detected.

```
(define-class Global-Variable Variable (initialized?))
(define (objectify-free-global-reference name r)
  (let ((v (make-Global-Variable name #f)))
    (set! g.current (cons v g.current))
    (make-Global-Reference v) ) )
```

Then insert the analysis in the compiler. It involves the interaction between the generic inspector and the generic function `inian!`.

```
(define (compile->C e out)
  (set! g.current '())
  (let ((prg (extract-things!
              (lift! (initialization-analyze! (Sexp->object e))) )))
    (gather-temporaries! (closurize-main! prg))
    (generate-C-program out e prg) ) )
(define (initialization-analyze! e)
  (call/cc (lambda (exit) (inian! e (lambda () (exit 'finished)))))
  e )
(define-generic (inian! (e) exit)
  (update-walk! inian! e exit) )
```

Now the problem is to follow the flow of computation and to determine the global variables that will surely be written before they are read. Rather than develop a specific and complicated analysis for that, why not try to determine a subset of variables that have surely been initialized? That's what the following five methods do.

```
(define-method (inian! (e Global-Assignment) exit)
  (call-next-method)
  (let ((gv (Global-Assignment-variable e)))
    (set-Global-Variable-initialized?! gv #t)
```

```
      (inian-warning "Surely initialized variable" gv)
      e ) )
  (define-method (inian! (e Global-Reference) exit)
    (let ((gv (Global-Reference-variable e)))
      (cond ((Predefined-Variable? gv) e)
            ((Global-Variable-initialized? gv) e)
            (else (inian-error "Surely uninitialized variable" gv)
                  (exit) ) ) ) )
  (define-method (inian! (e Alternative) exit)
    (inian! (Alternative-condition e) exit)
    (exit) )
  (define-method (inian! (e Application) exit)
    (call-next-method)
    (exit) )
  (define-method (inian! (e Function) exit)
    e )
```

That analysis follows the computation, determines assignments, and stops once the computation becomes too complicated to follow; that is, once a function is applied or once an alternative appears. In contrast, it is not necessary to look at the body of abstractions since a closure is calculated in finite time and without any errors.

Exercise 11.1 : The predicate `Object?` can be re-enforced and made less error-prone if you reserve another component in the vectors representing objects to contain a label. After creating this label, you should modify the predicate `Object?` as well as all the places where there is an allocation that should add this label, especially during bootstrapping, that is, when the predefined classes are defined.

```
(define *starting-offset* 2)
(define meroonet-tag (cons 'meroonet 'tag))
(define (Object? o)
  (and (vector? o)
       (>= (vector-length o) *starting-offset*)
       (eq? (vector-ref o 1) meroonet-tag) ) )
```

This modification will make the predicate more robust, but it does not increase the speed. However, the predicate can still be fooled if the label on an object is extracted by `vector-ref` and inserted in some other vector.

Exercise 11.2 : Since the function is generic, you can specialize it for certain classes. The following version gobbles up too many dotted pairs.

```
(define-generic (clone (o))
  (list->vector (vector->list o)) )
```

Exercise 11.3 : Define a new type of class, the metaclass `CountingClass`, with a supplementary field to count the allocations that will occur.

```
(define-class CountingClass Class (counter))
```

The code in MEROONET was written so that any modifications you make should not put everything else in jeopardy. It should be possible to define a class with that metaclass, like this:

```
(define-meroonet-macro (define-CountingClass name super-name
                                                own-field-descriptions )
  (let ((class (register-CountingClass
                name super-name own-field-descriptions )))
    (generate-related-names class) ) )
(define (register-CountingClass name super-name own-field-descriptions)
  (initialize! (allocate-CountingClass)
               name
               (->Class super-name)
               own-field-descriptions ) )
```

However, a better way of doing it would be to extend the syntax of `define-class` to accept an option indicating the metaclass to use, making `Class` the default. Then you must make several functions generic, like:

```
(define-generic (generate-related-names (class)))
(define-method (generate-related-names (class Class))
   (Class-generate-related-names class) )
(define-generic (initialize! (o) . args))
(define-method (initialize! (o Class) . args)
  (apply Class-initialize! o args) )
(define-method (initialize! (class CountingClass) . args)
  (set-CountingClass-counter! class 0)
  (call-next-method) )
```

The allocators in the accompanying functions should be modified to maintain the counter, like this:

```
(define-method (generate-related-names (class CountingClass))
  (let ((cname (symbol-append (Class-name class) '-class))
        (alloc-name (symbol-append 'allocate- (Class-name class)))
        (make-name (symbol-append 'make- (Class-name class))) )
    `(begin ,(call-next-method)
            (set! ,alloc-name          ;patch the allocator
                  (let ((old ,alloc-name))
                    (lambda sizes
                      (set-CountingClass-counter!
                       ,cname (+ 1 (CountingClass-counter ,cname)) )
                      (apply old sizes) ) ) )
            (set! ,make-name           ;patch the maker
                  (let ((old ,make-name))
                    (lambda args
                      (set-CountingClass-counter!
                       ,cname (+ 1 (CountingClass-counter ,cname)) )
                      (apply old args) ) ) ) ) ) )
```

To complete the exercise, here's an example of how to use it to count points:

```
(define-CountingClass CountedPoint Object (x y))
(unless (and (= 0 (CountingClass-counter CountedPoint-class))
             (allocate-CountedPoint)
             (= 1 (CountingClass-counter CountedPoint-class))
             (make-CountedPoint 11 22)
             (= 2 (CountingClass-counter CountedPoint-class)) )
```

```
;;should not be evaluated if everything is OK
(meroonet-error "Failed test on CountedPoint") )
```

Exercise 11.4 : Create a new metaclass, **ReflectiveClass**, with the supplementary fields, **predicate**, **allocator**, and **maker**. Then modify the way accompanying functions are generated so that these fields will be filled during the definition of the class. Do the same for the fields.

```
(define-class ReflectiveClass Class (predicate allocator maker))
(define-method (generate-related-names (class ReflectiveClass))
  (let ((cname (symbol-append (Class-name class) '-class))
        (predicate-name (symbol-append (Class-name class) '?))
        (allocator-name (symbol-append 'allocate- (Class-name class)))
        (maker-name (symbol-append 'make- (Class-name class))) )
    `(begin ,(call-next-method)
            (set-ReflectiveClass-predicate! ,cname ,predicate-name)
            (set-ReflectiveClass-allocator! ,cname ,allocator-name)
            (set-ReflectiveClass-maker! ,cname ,maker-name) ) ) )
```

Exercise 11.5 : The essential problem is to detect whether a generic function exists. In Scheme, it is not possible to know whether or not a global variable exists, so you'll have to consult the list ***generics***. Fortunately, it contains all the known generic functions.

```
(define-meroonet-macro (define-method call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((g (gensym))(c (gensym)))     ;make g and c hygienic
       `(begin
          (unless (->Generic ',(car call)) (define-generic ,call)) ;new
          (register-method
           ',(car call)
           (lambda (,g ,c)
             (lambda ,(flat-variables variables)
               (define (call-next-method)
                 ((if (Class-superclass ,c)
                      (vector-ref (Generic-dispatch-table ,g)
                                  (Class-number (Class-superclass ,c)) )
                      (Generic-default ,g) )
                  . ,(flat-variables variables) ) )
               . ,body ) )
           ',(cadr discriminant)
           ',(cdr call) ) ) ) ) ) )
```

Exercise 11.6 : Every method now has two local functions, **call-next-method** and **next-method?**. It would be clever not to generate them unless the body of the method contains calls to these functions.

```
(define-meroonet-macro (define-method call . body)
  (parse-variable-specifications
```

```
(cdr call)
(lambda (discriminant variables)
  (let ((g (gensym))(c (gensym)))     ;make g and c hygienic
    `(register-method
      ',(car call)
      (lambda (,g ,c)
        (lambda ,(flat-variables variables)
          ,@(generate-next-method-functions g c variables)
          . ,body ) )
      ',(cadr discriminant)
      ',(cdr call) ) ) ) ) )
```

The function **next-method?** is inspired by **call-next-method**, but it merely verifies whether a method exists; it doesn't bother to invoke one.

```
(define (generate-next-method-functions g c variables)
  (let ((get-next-method (gensym)))
    `((define (,get-next-method)
        (if (Class-superclass ,c)
            (vector-ref (Generic-dispatch-table ,g)
                        (Class-number (Class-superclass ,c)) )
            (Generic-default ,g) ) )
      (define (call-next-method)
        ((,get-next-method) . ,(flat-variables variables)) )
      (define (next-method?)
        (not (eq? (,get-next-method) (Generic-default ,g))) ) ) ) )
```

Bibliography

[85M85] *Macscheme Reference Manual.* Semantic Microsystems, Sausalito, California, 1985.

[All78] John Allen. *Anatomy of Lisp.* Computer Science Series. McGraw-Hill, 1978.

[ALQ95] Sophie Anglade, Jean-Jacques Lacrampe, and Christian Queinnec. Semantics of combinations in scheme. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 7(4):15–20, October–December 1995.

[App87] Andrew Appel. Garbage Collection can be faster than Stack Allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[App92a] Andrew Appel. *Compiling with continuations.* Cambridge Press, 1992.

[App92b] Apple Computer, Eastern Research and Technology. *Dylan, An object-oriented dynamic language.* Apple Computer, Inc., April 1992.

[AR88] Norman Adams and Jonathan Rees. Object-oriented programming in Scheme. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 277–288, August 1988.

[AS85] Harold Abelson and Gerald Jay with Julie Sussman Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

[AS94] Andrew W Appel and Zhong Shao. An empirical and analytic study of stack vs heap cost for languages with closures. Technical Report CS-TR-450-94, Princeton University, Department of Computer Science, Princeton (New-Jersey USA), March 1994.

[Att95] Giuseppe Attardi. The embeddable common Lisp. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 8(1):30–41, January–April 1995. LUV-94, Fourth International LISP Users and Vendors Conference.

[Bak] Henry G Baker. Cons should not cons its arguments, part ii: Cheney on the m.t.a. `ftp://ftp.netcom.com/pub/hb/hbaker/CheneyMTA.ps`.

[Bak78] Henry G Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bak92a] Henry G Baker. The buried binding and dead binding problems of Lisp 1.5. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 5(2):11–19, April–June 1992.

[Bak92b] Henry G Baker. Inlining semantics for subroutines which are recursive. *SIGPLAN Notices*, 27(12):39–46, December 1992.

[Bak92c] Henry G Baker. Metacircular semantics for common Lisp special forms. *Lisp Pointers*, 5(4):11–20, 1992.

[Bak93] Henry G Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4):2–27, October 1993.

[Bar84] H P Barendregt. *The Lambda Calculus.* Number 103 in Studies in Logic. North Holland, Amsterdam, 1984.

[Bar89] Joel F. Bartlett. Scheme->c a portable scheme-to-c compiler. Research Report 89 1, DEC Western Research Laboratory, Palo Alto, California, January 1989.

[Baw88] Alan Bawden. Reification without evaluation. In *Conference Record of the 1988 ACM Symposium on LISP and Functional Programming*, pages 342–351, Snowbird, Utah, July 1988. ACM Press.

[BC87] Jean-Pierre Briot and Pierre Cointe. A uniform model for object-oriented languages using the class abstraction. In *IJCAI '87*, pages 40–43, 1987.

[BCSJ86] Jean-Pierre Briot, Pierre Cointe, and Emmanuel Saint-James. Réécriture et récursion dans une fermeture, étude dans un Lisp à liaison superficielle et application aux objets. In Pierre Cointe and Jean Bézivin, editors, $3^{èmes}$ *journées LOO/AFCET*, number 48 in Revue Bigre+Globule, pages 90–100, IRCAM, Paris (France), January 1986.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification. *SIGPLAN Notices*, 23, September 1988. special issue.

[Bel73] James R Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, June 1973.

[Bet91] David Michael Betz. *XSCHEME: An Object-oriented Scheme.* P.O. Box 144, Peterborough, NH 03458 (USA), July 1991. version 0.28.

[BG94] Henri E Bal and Dick Grune. *Programming Language Essentials.* Addison Wesley, 1994.

[BHY87] Adrienne Bloss, Paul Hudak, and Jonathan Young. Code optimizations for lazy evaluation. *International journal on Lisp and Symbolic Computation*, 1(2):147–164, 1987.

[BJ86] David H Bartley and John C Jensen. The implementation of pc Scheme. In *Conference Record of the 1986 ACM Symposium on LISP and Functional Programming*, pages 86–93, Cambridge, Massachusetts, August 1986. ACM Press.

[BKK+86] D G Bobrow, K Kahn, G Kiczales, L Masinter, M Stefik, and F Zdybel. Commonloops: Merging Lisp and object oriented programming. In *OOPSLA '86 — Object-Oriented Programming Systems and LAnguages*, pages 17–29, Portland (Oregon, USA), 1986.

[BM82] Robert S. Boyer and J. Strother Moore. A mechanical proof of the unsolvability of the halting problem. Technical Report ICSCA-CMP-28, July 1982.

[BR88] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on LISP and Functional Programming*, Salt Lake City, Utah., July 1988.

[BW88] Hans J Boehm and M Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9), September 1988.

[Car93] Luca Cardelli. An implementation of $F_{<:}$. Research Report 97, DEC-SRC, February 1993.

[Car94] George J. Carrette. *Siod, Scheme In One Defun.* PARADIGM Associates Incorporated, 29 Putnam Ave, Suite 6, Cambridge, MA 02138, USA, 1994.

[Cay83] Michel Cayrol. *Le Langage Lisp.* Cepadues Editions, Toulouse (France), 1983.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77 — 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977. ACM Press.

[CDD+91] J. Chailloux, M. Devin, F. Dupont, J.-M. Hullot, B. Serpette, and J. Vuillemin. *Le-Lisp version 15.24, le manuel de référence.* INRIA, May 1991.

[CEK+89] L W Cannon, R A Elliott, L W Kirchhoff, J H Miller, J M Milner, R W Mitze, E P Schan, N O Whittington, H Spencer, and D Keppel. *Recommended C Style and Coding Standards*, November 1989.

[CG77] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in LISP. *Communications of the ACM*, 20(2):78–87, February 1977.

[CH94] William D Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming* [lfp94], pages 128–139.

[Cha80] Jérôme Chailloux. *Le modèle VLISP : description, implémentation et évaluation.* Thèse de troisième cycle, Université de Vincennes, April 1980. Rapport LITP 80-20, Paris (France).

[Cha94] Gregory J Chaitin. The limits of mathematics. IBM PO Box 704, Yorktown Heights, NY 10598 (USA), July 1994.

[CHO88] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 124–131, August 1988.

[Chr95] Christophe. *La Famille Fenouillard.* Armand Colin, 1895.

[Cla79] Douglas W. Clark. Measurements of dynamic list structure use in Lisp. *IEEE Transactions on Software Engineering*, 5(1):51–59, January 1979.

[Cli84] William Clinger. The Scheme 311 compiler: an exercise in denotational semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, 1984.

[Coi87] Pierre Cointe. The ObjVlisp kernel: a reflexive architecture to define a uniform object oriented system. In P. Maes and D. Nardi, editors, *Workshop on MetaLevel Architectures and Reflection*, Alghiero, Sardinia (Italy), October 1987. North Holland.

[Com84] Douglas Comer. *Operating System Design: The XINU Approach.* Prentice-Hall, 1984.

[CR91a] William Clinger and Jonathan Rees. Macros that work. In *POPL '91 – Eighteenth Annual ACM symposium on Principles of Programming Languages*, pages 155–162, Orlando, (Florida USA), January 1991.

[CR91b] William Clinger and Jonathan A Rees. The revised[4] report on the algorithmic language Scheme. *Lisp Pointer*, 4(3), 1991.

[Cur89] Pavel Curtis. (algorithms). *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 3(1):48–61, July 1989.

[Dan87] Olivier Danvy. Memory allocation and higher-order functions. In *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, SIGPLAN Notices, Vol. 22, No 7, pages 241–252, Saint-Paul, Minnesota, June 1987. ACM, ACM Press.

[Del89] Vincent Delacour. Picolo expresso. *Revue Bigre+Globule*, (65):30–42, July 1989.

[Deu80] L Peter Deutsch. Bytelisp and its alto implementation. In *Conference Record of the 1980 LISP Conference* [lfp80], pages 231–242.

[Deu89] Alain Deutsch. Génération automatique d'interprèteurs et compilation à partir de spécifications dénotationnelles. Rapport de DEA-LAP 1988 LITP-RXF 89-17, LITP, January 1989.

[Dev85] Matthieu Devin. Le portage du système Le-Lisp. Rapport technique 50, INRIA-Rocquencourt, May 1985.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 151–160, Nice (France), June 1990.

[DFH86] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: Beyond conventional macros. In *Conference Record of the 1986 ACM Conference on Lisp and Functional Programming*, pages 143–150, 1986.

[DFH88] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: a general macro mechanism. *International journal on Lisp and Symbolic Computation*, 1(1):53–76, June 1988.

[DH92] Olivier Danvy and John Hatcliff. Thunks (continued). In *Proceedings of the Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 3–11, Bordeaux, France, September 1992. IRISA, Rennes, France. Extended version available as Technical Report CIS-92-28, Kansas State University.

[DHB93] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *International journal on Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[DHHM92] R Ducournau, M Habib, M Huchard, and M L Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *OOPSLA '92 — Object-Oriented Programming Systems and LAnguages*, pages 16–4, 1992.

[Dil88] Antoni Diller. *Compiling Functional Languages*. John Wiley and sons, 1988.

[DM88] Olivier Danvy and Karoline Malmkjær. A Blond primer. DIKU report 88/21, University of Copenhagen, Copenhagen, Denmark, 1988.

[dM95] Antoine Dumesnil de Maricourt. *Macro-expansion en Lisp, sémantique et réalisation*. Thèse d'université, Université Paris 7, Paris (France), June 1995.

[DPS94a] Harley Davis, Pierre Parquier, and Nitsan Séniak. Sweet harmony: the talk/c++ connection. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming* [lfp94], pages 121–127.

[DPS94b] Harley Davis, Pierre Parquier, and Nitsan Séniak. Talking about modules and delivery. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming* [lfp94], pages 113–120.

[dR87] Jim des Rivières. Control-related meta-level facilities in Lisp. In P. Maes and D. Nardi, editors, *Workshop on Meta-Level Architecture and Reflection*, Alghiero, Sardinia (Italy), October 1987. North Holland.

[dRS84] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 331–347, Austin, Texas, August 1984. ACM Press.

[Dyb87] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[ES70] Jay Earley and Howard Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10):607–617, October 1970.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *POPL '88 – Fifteenth Annual ACM symposium on Principles of Programming Languages*, pages 180–190, San Diego (California USA), January 1988.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. In Neil Jones, editor, *ESOP '90 – European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 134–151, Copehaguen (Danmark), 1990. Springer-Verlag.

[FF87] Matthias Felleisen and Daniel P. Friedman. A reduction semantics for imperative higher-order languages. *Parallel Architectures and Languages Europe*, 259:206–223, 1987.

[FF89] Matthias Felleisen and Daniel P Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989. Preliminary version in: *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987, 314-325.

[FFDM87] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Computer Science Dept. Technical Report 216, Indiana University, Bloomington, Indiana, February 1987.

[FH89] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. Computer Science Technical Report No. 100, Rice University, June 1989.

[FL87] Marc Feeley and Guy LaPalme. Using closures for code generation. *Journal of Computer Languages*, 12(1):47–66, 1987.

[FM90] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[FW76] D.P. Friedman and D.S. Wise. Cons should not evaluate its arguments. In Michaelson and Milner, editors, *Proceedings of the 3rd International Colloquium on "Automata, Languages and Programming"*, pages 257–284. Edinburgh University Press, July 1976.

[FW84] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 348–355, Austin, TX., August 1984.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce Duba. Abstract continuations: a mathematical semantics for handling functional jumps. In *Proceedings of the 1988 ACM Symposium on LISP and Functional Programming*, Salt Lake City, Utah., July 1988.

[FWH92] Daniel P Friedman, Mitchell Wand, and Christopher Haynes. *Essentials of Programming Languages.* MIT Press, Cambridge MA and McGraw-Hill, 1992.

[Gab88] Richard P Gabriel. The why of y. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 2(2):15–25, 1988.

[GBM82] Martin L. Griss, Eric Benson, and Gerald Q Maguire. Psl: A portable LISP system. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 88–97, Pittsburgh, Pennsylvania, August 1982. ACM Press.

[Gor75] Michael J C Gordon. Operational reasoning and denotational semantics. In G. Huet and G. Kahn, editors, *Actes du Colloque IRIA "Constructions et Justifications de Programmes"*, pages 83–98, Arc et Senans, July 1975.

[Gor88] Michael J C Gordon. *Programming Language Theory and its Implementation.* International Series in Computer Science. Prentice-Hall, 1988.

[GP88] Richard P Gabriel and Kent M Pitman. Technical issues of separation in function cells and value cells. *International journal on Lisp and Symbolic Computation*, 1(1):81–101, June 1988.

[GR83] Adèle Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison Wesley, 1983.

[Gra93] Paul Graham. *On Lisp, Advanced Techniques for Common Lisp.* Prentice-Hall, 1993.

[Gre77] Patrick Greussay. *Contribution à la définition interprétative et à l'implémentation des Lambda-langages.* Thèse d'état, Université Paris VI, November 1977. Rapport LITP 78-2.

[Gud93] David Gudeman. Representing type information in dynamically typed languages. Technical Report 93–27, Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, October 1993.

[Han90] Chris Hanson. Efficient stack allocation for tail-recursive languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[HD90] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *PPOPP '90 – ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 128–136, Seattle (Washington US), March 1990.

[HDB90] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, White Plains, New York, June 1990.

[Hen80] Peter Henderson. *Functional Programming, Application and Implementation.* International Series in Computer Science. Prentice-Hall, 1980.

[Hen92a] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, San Francisco, USA, June 1992.

[Hen92b] Wade Hennessey. Wcl: Delivering efficient COMMON LISP applications under UN*X. In *Conference Record of the 1992 ACM Symposium on LISP and Functional Programming*, pages 260–269, San Francisco, California, June 1992. ACM Press.

[HFW84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, TX., 1984.

[Hof93] Ulrich Hoffmann. Using c as target code for translating highlevel programming languages. Technical Report APPLY/CAU/IV.4/2, Christian-Albrechts-Universitey of Kiel, 1993.

[Hon93] P. Joseph Hong. Threaded code designs for forth interpreters. *SIG FORTH, Newsletter of the ACM's Special Interest Group on FORTH*, 4(2):11–18, fall 1993.

[HS75] Carl Hewitt and Brian Smith. Towards a programming apprentice. *IEEE Transactions on Software Engineering*, 1(1):26–45, March 1975.

[HS91] Samuel P Harbison and Guy L Steele, Jr. *C: A Reference Manual.* Prentice-Hall, 1991.

[IEE91] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[ILO94] ILOG, 2 Avenue Galliéni, BP 85, 94253 Gentilly, France. *Ilog-Talk Reference Manual*, 1994.

[IM89] Takayasu Ito and Manabu Matsui. A parallel lisp language PaiLisp and its kernel specification. In Takayasu Ito and Robert H Halstead, Jr., editors, *Proceedings of the US/Japan Workshop on Parallel Lisp*, volume Lecture Notes in Computer Science 441, pages 58–100, Sendai (Japan), June 1989. Springer-Verlag.

[IMY92] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCL: A reflective object-oriented concurrent language without a run-time kernel. In Yonezawa and Smith [YS92], pages 24–35.

[ISO90] ISO/IEC 9899:1990. Programmming language — c. Technical report, Information Technology, 1990.

[ISO94] ISO-IEC/JTC1/SC22/WG16. Programming language ISLISP, CD 13816. Technical report, ISO-IEC/JTC1/SC22/WG16, 1994.

[ITW86] Takayasu Ito, Takashi Tamura, and Shin-ichi Wada. Theoretical comparisons of interpreted/compiled executions of Lisp on sequential and parallel machine models. In H J Kugler, editor, *Proceedings of the IFIP 10th World Computer Congress*, Dublin (Ireland), September 1986. North Holland.

[Jaf94] Aubrey Jaffer. *Reference Manual for scm*, 1994.

[JF92] Stanley Jefferson and Daniel P Friedman. A simple reflective interpreter. In Yonezawa and Smith [YS92], pages 48–58.

[JGS93] Neil D Jones, C Karsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993. With chapters by L.O. Andersen and T. Mogensen.

[Kah87] Gilles Kahn. Natural semantics. In *Proc. of STACS 1987, Lecture Notes in Computer Science*, chapter 247. Springer–Verlag, March 1987.

[Kam90] Samuel Kamin. *Programming Languages: an Interpreter-Based Approach.* Addison-Wesley, Reading, Mass., 1990.

[KdRB92] Gregor Kiczales, Jim des Rivières, and Daniel G Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge MA, 1992.

[Kes88] Robert R. Kessler. *Lisp, Objects, and Symbolic Programming.* Scott, Foreman/Little, Brown College Division, Glenview, Illinois, 1988.

[KFFD86] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. *Symposium on LISP and Functional Programming*, pages 151–161, August 1986.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *POPL'89 — 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 281–292, Austin (Texas, USA), January 1989. ACM Press.

[Knu84] Donald Ervin Knuth. *The TEX Book.* Addison Wesley, 1984.

[KR78] Brian W Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in pcl. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 99–1052, Nice (France), June 1990.

[KW90] Morry Katz and Daniel Weise. Continuing into the future: on the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[Lak80] Fred H. Lakin. Computing with text-graphic forms. In *Conference Record of the 1980 Lisp Conference*, pages 100–105, 1980.

[Lan65] P J Landin. A Correspondence between algol 60 and Church's Lambda-notation. *Communications of the ACM*, 8:89–101 and 158–165, February 1965.

[Leb05] Maurice Leblanc. *813.* Éditions Pierre Lafitte, 1905.

[LF88] Henry Lieberman and Christopher Fry. Common eval. *Lisp Pointers*, 2(1):23–33, July 1988.

[LF93] Shinn-Der Lee and Daniel P Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *POPL '93 – Twentieth Annual ACM symposium on Principles of Programming Languages*, pages 479–492, Charleston (South Carolina, USA), January 1993. ACM Press.

[lfp80] *Conference Record of the 1980 LISP Conference*, Stanford (California USA), August 1980. The LISP Conference.

[lfp94] *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando (Florida USA), June 1994. ACM Press.

[Lie87] Henry Lieberman. Reversible object-oriented interpreters. In *ECOOP '87 – European Conference on Object-Oriented Programming*, volume Special issue of Bigre 54, pages 13–22, Paris (France), June 1987. AFCET.

[LLSt93] Bil Lewis, Dan LaLiberte, Richard Stallman, and the GNU Manual Group. Gnu emacs Lisp reference manual. Technical report, Free Software Foundation, 675 Massachusetts Avenue, Cambridge MA 02139 USA, Edition 2.0 1993.

[LP86] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp: an object-oriented Scheme with first class types. In *ACM Conference on Object-Oriented Systems, Programming, Languages and Applications*, pages 30–37, September 1986.

[LP88] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp: an object-oriented dialect of Scheme. *International journal on Lisp and Symbolic Computation*, 1(1):39–51, May 1988.

[LW93] Xavier Leroy and Pierre Weis. *Manuel de référence du langage Caml.* InterÉditions, 1993.

[MAE+62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. Lisp 1.5 programmer's manual. Technical report, MIT Press, Cambridge, MA (USA), 1962.

[Man74] Zohar Manna. *Mathematical Theory of Computation.* Computer Science Series. McGraw-Hill, 1974.

[Mas86] Ian A Mason. *The Semantics of Destructive Lisp.* CSLI Lecture Notes, Leland Stanford Junior University, California USA, 1986.

[Mat92] Luis Mateu. Efficient implementation of coroutines. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 230–247, Saint-Malo (France), September 1992. Springer-Verlag.

[MB93] Luis Mateu-Brule. *Stratégies avancées de gestion de blocs de contrôle.* Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris 6), Paris (France), February 1993.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine – part i. *Communications of the ACM*, 3(1):184–195, 1960.

[McC78a] John McCarthy. Lisp history. In *Proc. SIGPLAN History of Programming Languages Conference*, pages 217–223, 1978. Also Sigplan Notices 13(8).

[McC78b] John McCarthy. A micro-manual for Lisp - not the whole truth. In *Proc. SIGPLAN History of Programming Languages Conference*, pages 215–216, 1978. Also Sigplan Notices 13(8).

[McD93] Raymond C McDowell. The relatedness and comparative utility of various approaches to operational semantics. Technical Report MS-CIS-93-16, LINC LAB 246, University of Pennsylvania, February 1993.

[MNC+89] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, and Karl Tombre. *Les langages à objets.* InterÉditions, 1989.

[Mor92] Luc Moreau. An operational semantics for a parallel functional language with continuations. In D. Etiemble and J-C. Syre, editors, *PARLE '92 – Parallel Architectures and Languages Europe*, pages 415–430, Paris (France), June 1992. Lecture Notes in Computer Science 605, Springer-Verlag.

[Mor94] Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations.* Thèse de docteur en sciences appliquées, Université de Liège (Belgique), avril 1994.

[Mos70] Joel Moses. The function of FUNCTION in LISP. *SIGSAM Bulletin*, 15:13–27, July 1970.

[Moz87] Wolfgang A Mozart. Don Giovanni, 1787.

[MP80] Steven S. Muchnick and Uwe F. Pleban. A semantic comparison of lisp and Scheme. In *Conference Record of the 1980 Lisp Conference*, pages 56–65. The Lisp Conference, P.O. Box 487, Redwood Estates CA., 1980.

[MQ94] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *Lecture Notes in Computer Science 844*, pages 182–197, Madrid (Spain), September 1994. International Workshop PLILP '94 – Programming Language Implementation and Logic Programming, Springer-Verlag.

[MR91] James S Miller and Guillermo J Rozas. Free variables and first-class environments. *International journal on Lisp and Symbolic Computation*, 4(2):107–141, 1991.

[MS80] F Lockwood Morris and Jerald S Schwarz. Computing cyclic list structures. In *Conference Record of the 1980 Lisp Conference*, pages 144–153. The Lisp Conference, 1980.

[Mul92] Robert Muller. M-lisp: A representation-independent dialect of lisp with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14(4):589–615, October 1992.

[Nei84] Eugen Neidl. *Étude des relations avec l'interprète dans la compilation de Lisp.* Thèse de troisième cycle, Université Paris 6, Paris (France), 1984.

[Nor72] Eric Norman. 1100 Lisp reference manual. Technical report, University of Wisconsin, 1972.

[NQ89] Greg Nuyens and Christian Queinnec. Identifier Semantics: a Matter of References. Technical Report LIX RR 89 02, 67–80, Laboratoire d'Informatique de l'École Polytechnique, May 1989.

[PE92] Julian Padget and Greg Nuyens (Editors). The eulisp definition. Technical report, University of Bath, 1992.

[Per79] Jean-François Perrot. Lisp et λ-calcul. In Bernard Robinet (ed), editor, *λ-calcul et sémantique formelle des langages de programmation*, La Châtre (France), 1979. AFCET-GROPLAN, LITP-ENSTA, Paris.

[Pit80] Kent Pitman. Special forms in Lisp. In *Conference Record of the 1980 LISP Conference* [lfp80], pages 179–187.

[PJ87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages.* International Series in Computer Science. Prentice-Hall, 1987.

[PNB93] Padget, J.A., Nuyens, G., and Bretthauer, H. An overview of EuLisp. *Lisp and Symbolic Computation*, 6(1/2):9–98, 1993.

[QC88] Christian Queinnec and Pierre Cointe. An open-ended Data Representation Model for Eu-Lisp. In *LFP '88 - ACM Symposium on Lisp and Functional Programming*, pages 298–308, Snowbird (Utah, USA), 1988.

[QD93] Christian Queinnec and David De Roure. Design of a concurrent and distributed language. In Robert H Halstead Jr and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, volume Lecture Notes in Computer Science 748, pages 234–259, Boston (Massachussetts USA), October 1993.

[QD96] Christian Queinnec and David De Roure. Sharing code through first-class environments. In *Proceedings of ICFP'96 — ACM SIGPLAN International Conference on Functional Programming*, pages ??–??, Philadelphia (Pennsylvania, USA), May 1996.

[QG92] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to symbolic pattern matching with intelligent backtrack. In M Billaud, P Castéran, MM Corsini, K Musumbu, and A Rauzy, editors, *WSA '92—Workshop on Static Analysis*, number 81-82 in Revue Bigre+Globule, pages 109–117, Bordeaux (France), September 1992.

[QP90] Christian Queinnec and Julian Padget. A deterministic model for modules and macros. Bath Computing Group Technical Report 90-36, University of Bath, Bath (UK), 1990.

[QP91a] Christian Queinnec and Julian Padget. A proposal for a modular Lisp with macros and dynamic evaluation. In *Journées de Travail sur l'Analyse Statique en Programmation Équationnelle, Fonctionnelle et Logique*, pages 1–8, Bordeaux (France), October 1991. Revue Bigre+Globule 74.

[QP91b] Christian Queinnec and Julian Padget. Modules, macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[QS91] Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *POPL '91 – Eighteenth Annual ACM symposium on Principles of Programming Languages*, pages 174–184, Orlando (Florida USA), January 1991.

[Que82] Christian Queinnec. *Langage d'un autre type : Lisp.* Eyrolles, Paris (France), 1982.

[Que89] Christian Queinnec. Lisp – Almost a whole Truth. Technical Report LIX/RR/89/03, 79–106, Laboratoire d'Informatique de l'École Polytechnique, December 1989.

[Que90a] Christian Queinnec. A Framework for Data Aggregates. In Pierre Cointe, Philippe Gautron, and Christian Queinnec, editors, *Actes des JFLA 90 – Journées Francophones des Langages Applicatifs*, pages 21–32, La Rochelle (France), January 1990. Revue Bigre+Globule 69.

[Que90b] Christian Queinnec. *Le filtrage : une application de (et pour) Lisp.* InterÉditions, Paris (France), 1990. ISBN 2-7296-0332-8.

[Que90c] Christian Queinnec. PolyScheme : A Semantics for a Concurrent Scheme. In *Workshop on High Performance and Parallel Computing in Lisp*, Twickenham (UK), November 1990. European Conference on Lisp and its Practical Applications.

[Que92a] Christian Queinnec. Compiling syntactically recursive programs. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 5(4):2–10, October-December 1992.

[Que92b] Christian Queinnec. A concurrent and distributed extension to scheme. In D. Etiemble and J-C. Syre, editors, *PARLE '92 – Parallel Architectures and Languages Europe*, pages 431–446, Paris (France), June 1992. Lecture Notes in Computer Science 605, Springer-Verlag.

[Que93a] Christian Queinnec. Continuation conscious compilation. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 6(1):2–14, January 1993.

[Que93b] Christian Queinnec. Designing MEROON v3. In Christian Rathke, Jürgen Kopp, Hubertus Hohl, and Harry Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany), September 1993.

[Que93c] Christian Queinnec. A library of high-level control operators. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 6(4):11–26, October 1993.

[Que93d] Christian Queinnec. Literate programming from scheme to TEX. Research Report LIX RR 93.05, Laboratoire d'Informatique de l'École Polytechnique, 91128 Palaiseau Cedex, France, November 1993.

[Que94] Christian Queinnec. Locality, causality and continuations. In *LFP '94 – ACM Symposium on Lisp and Functional Programming*, pages 91–102, Orlando (Florida, USA), June 1994. ACM Press.

[Que95] Christian Queinnec. Dmeroon overview of a distributed class-based causally-coherent data model. In T. Ito, R. Halstead, and C. Queinnec, editors, *PSLS 95 – Parallel Symbolic Langages and Systems*, Beaune (France), October 1995.

[R3R86] Revised[3] report on the algorithmic language scheme. *ACM Sigplan Notices*, 21(12), December 1986.

[RA82] Jonathan A. Rees and Norman I. Adams. T: a dialect of lisp or, lambda: the ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122, 1982.

[RAM84] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual, Fourth Edition.* Yale University Computer Science Department, January 1984.

[Ray91] Eric Raymond. *The New Hacker's Dictionary.* MIT Press, Cambridge MA, 1991. With assistance and illustrations by Guy L. Steele Jr.

[Rey72] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[Rib69] Daniel Ribbens. *Programmation non numérique : Lisp 1.5.* Monographies d'Informatique, AFCET, Dunod, Paris, 1969.

[RM92] John R Rose and Hans Muller. Integrating the Scheme and c languages. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 247–259, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[Row90] William Rowan. A Lisp compiler producing compact code. In *Conference Record of the 1980 Lisp Conference* [lfp80], pages 216–222.

[Roz92] Guillermo Juan Rozas. Taming the y operator. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 226–234, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[Sam79] Hanan Sammet. Deep and shallow binding: The assignment operation. *Computer Languages*, 4:187–198, 1979.

[Sch86] David A Schmidt. *Denotational Semantics, a Methodology for Language Development.* Allyn and Bacon, 1986.

[Sco76] Dana Scott. Data types as lattices. *Siam J. Computing*, 5(3):522–587, 1976.

[Sén89] Nitsan Séniak. Compilation de Scheme par spécialisation explicite. *Revue Bigre+Globule*, (65):160–170, July 1989.

[Sén91] Nitsan Séniak. *Théorie et pratique de Sqil, un langage intermédiaire pour la compilation des langages fonctionnels.* Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris 6), Paris (France), October 1991.

[Ser93] Manuel Serrano. De l'utilisation des analyses de flot de contrôle dans la compilation des langages fonctionnels. In Pierre Lescanne, editor, *Actes des journées du GDR de Programmation*, 1993.

[Ser94] Manuel Serrano. *Bigloo User's Manual*, 1994. available on `ftp://ftp.inria.fr/INRIA/Projects/icsla/Implementations`.

[SF89] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming.* MIT Press and McGraw-Hill, 1989.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about continuation-passing style programs. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 288–298, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[SG93] Guy L. Steele, Jr. and Richard P Gabriel. The evolution of Lisp. In *The Second ACM SIGPLAN History of Proramming Languages Conference (HOPL-II)*, pages 231–270, Cambridge (Massachusetts, USA), April 1993. ACM SIGPLAN Notices 8, 3.

[Shi91] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation.* The MIT Press, Cambridge, MASS, 1991.

[SJ87] Emmanuel Saint-James. *De la Méta-Récursivité comme Outil d'Implémentation.* Thèse d'état, Université Paris VI, December 1987.

[SJ93] Emmanuel Saint-James. *La programmation appplicative (de LISP à la machine en passant par le lambda-calcul).* Hermès, 1993.

[Sla61] J R Slagle. *A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculees, Symbolic Automatic Integration (SAINT).* PhD thesis, MIT, Lincoln Lab, 1961.

[Spi90] Éric Spir. *Gestion dynamique de la mémoire dans les langages de programmation, application à Lisp.* Science informatique. InterÉditions, Paris (France), 1990.

[SS75] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT AI Memo 349, Massachusetts Institute of Technology, Cambridge, Mass., December 1975.

[SS78a] Guy Lewis Steele Jr. and Gerald Jay Sussman. The art of the interpreter, or the modularity complex (parts zero, one, and two). MIT AI Memo 453, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[SS78b] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of lisp. MIT AI Memo 452, Massachusetts Institute of Technology, Cambridge, Mass., January 1978.

[SS80] Guy Lewis Steele Jr. and Gerald Jay Sussman. The dream of a lifetime: a lazy variable extent mechanism. In *Conference Record of the 1980 Lisp Conference*, pages 163–172. The Lisp Conference, 1980.

[Ste78] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[Ste84] Guy L. Steele, Jr. *Common Lisp, the Language.* Digital Press, Burlington MA (USA), 1984.

[Ste90] Guy L. Steele, Jr. *Common Lisp, the Language.* Digital Press, Burlington MA (USA), 2nd edition, 1990.

[Sto77] Joseph E Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge Massachussetts USA, 1977.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1986. ou "Le langage C++", InterÉditions 1989.

[SW94] Manuel Serrano and Pierre Weis. 1+1=1: An optimizing caml compiler. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 101–111, Orlando (Florida, USA), June 1994. INRIA RR 2265.

[Tak88] M. Takeichi. Lambda-hoisting: A transformation technique for fully lazy evaluation of functional programs. *New Generation Computing*, (5):377–391, 1988.

[Tei74] Warren Teitelman. *InterLISP Reference Manual.* Xerox Palo Alto Research Center, Palo Alto (California, USA), 1974.

[Tei76] Warren Teitelman. Clisp: Conversational LISP. *IEEE Trans. on Computers*, C-25(4):354–357, April 1976.

[VH94] Jan Vitek and R Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In *ECOOP '94 — 8th European Conference on Object-Oriented Programming*, Bologna (Italy), 1994.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In H Ganzinger, editor, *ESOP '88 – European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358, 1988.

[Wan80a] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28. The Lisp Conference, 1980.

[Wan80b] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.

[Wan84] Mitchell Wand. A semantic prototyping system. In *Proceedings ACM SIGPLAN '84 Compiler Construction Conference*, pages 213–221, 1984.

[Wan86] Mitchell Wand. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Symposium on LISP and Functional Programming*, pages 298–307, August 1986.

[Wat93] Richard C Waters. Macroexpand-all: An example of a simple Lisp code walker. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 6(1):25–32, January 1993.

[WC94] Andrew K Wright and Robert Cartwright. A practical soft typing system for Scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming* [lfp94], pages 250–262.

[WH88] Patrick H Winston and Berthold K Horn. *Lisp.* Addison Wesley, third edition, 1988.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[WL93] Pierre Weis and Xavier Leroy. *Le langage Caml.* InterÉditions, 1993.

[WS90] Larry Wall and Randal L Schwartz. *Programming perl.* O'Reilly & Associates, Inc., 1990.

[WS94] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *POPL'94 — 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 435–445, Portland (Oregon, USA), January 1994. ACM Press.

[YH85] Taiichi Yuasa and Masami Hagiya. Kyoto common Lisp report. Technical report, Kyoto University, Japan, 1985.

[YS92] Akinori Yonezawa and Brian C. Smith, editors. *Reflection and Meta-Level Architecture: Proceedings of the International Workshop on New Models for Software Architecture '92*, Tokyo, Japan, November 1992. Research Institute of Software Engineering (RISE) and Information-Technology Promotion Agency, Japan (IPA), in cooperation with ACM SIGPLAN, JSSST, IPSJ.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X