when it is necessary to provide an explanation, or even temporarily, to track a difficult-to-catch bug.

Exercise 6.5: Justifications

Sketch out the issues involved in carrying justifications for data. Notice that the reason for a value depends on the values that it was derived from and the way those values were combined. What do we do if the reason for a value is some numerically weighted combination of many factors, as in a deep neural network? This is a research question that we need to address to make the systems that affect us accountable.

6.5 Implementing layers: a second pass.

The layering idea seems to be good. The implementation (under MIT/GNU Scheme) shown earlier (section 6.2) is sufficient to illustrate the ideas and to support dependency tracking for the propagator system to be developed in Chapter 7. But the implementation has serious problems.

- Conditionals are not explicitly handled in the implementation.
  - The support of the value of an `if` expression must include the support of the value of the predicate expression and the support of the value of the selected consequent expression or alternative expression.
- The implementation breaks tail-call optimization.
  - The result of applying a layered procedure is a new layered data item that must be put together from the values of the component layers of the layered procedure, *after* those components return their values. Thus, tail calls in a layered procedure's layers cannot be optimized.

These problems cannot be easily addressed in the underlying Scheme. To make conditionals sensitive to a layered datum as the value of the predicate expression would require changes to the underlying interpreter and compiler, or perhaps some magic macro that changes the meaning of `if`: the consequent or alternative expression chosen by the base value of the predicate expression would then need its (perhaps layered) value merged with the layers

of the value of the predicate expression, also breaking tail-call optimization.

However, it is possible to make an interpreter (and thus a compiler) that solves these problems in a way that correctly implements `if` and respects tail-call optimization. The key observation is that the continuations (whose job is waiting for a value from a subexpression evaluation) can be enhanced to collect the results from multiple layers, and combine them to make a layered datum after all of the layer contributions have been accumulated.

Using the same mechanism, a continuation waiting for the value of an `if` expression can accept and hold informtion developed when evaluating the predicate and combine that information with the value of the chosen consequent or alternative.

6.5.1 Interpreter with extended continuations

Some Scheme systems [35] have extended continuations to allow the implementation of continuation marks [94], which allow continuations to be decorated with key-value pairs. The continuation mark mechanism can be used to implement exactly what is needed to allow the accumulation of results of multiple layers without breaking tail-call optimization. But this does not, in itself, solve the `if` problem. So, to make things more clear we will provide an entire interpreter that has extended continuations and also allows `if` to combine layers from the predicate with the layers from the selected consequent or alternative.

We start with the interpreter of section 5.5.4. This interpreter was written in continuation-passing style, with explicit continuations that could be provided to the interpreted program using `call/cc`. Here we will extend each continuation that appears in this interpreter so that it can do more than just receive a value for a subexpression: it will support a continuation-marks-like feature.

The new version of `eval` is mostly the same:[7]

[7] In this evaluator we use the `l:` prefix (for *layered*) to distinguish analogous procedures, as explained in footnote 19 on page 260.

```
(define (l:eval expression environment continue)
  ((analyze expression) environment continue))

(define (analyze expression)
  (make-executor (c:analyze expression)))

(define (default-analyze expression)
  (cond ((application? expression)
         (analyze-application expression))
        (else (error "Unknown expression type" expression))))

(define l:analyze
  (simple-generic-procedure 'l:analyze 1 default-analyze))
```

As before the result of `(analyze expression)` is an execution procedure that takes an environment and a continuation procedure.

The first significant change is in `analyze-application`. The bare continuation `(lambda (proc) ...)` is wrapped with the procedure `extend-continuation`:

```
(define (analyze-application expression)
  (let ((operator-exec (analyze (operator expression)))
        (operand-execs (map analyze (operands expression))))
    (lambda (environment continue)
      (l:execute-strict operator-exec environment
        (extend-continuation
          (lambda (proc) ; The continuation to be extended
            (l:apply proc
                     operand-execs
                     environment
                     continue)))))))
```

The procedure `extend-continuation` produces an extended continuation that is just its argument, enhanced with the key-value association mechanism that is part of the continuation marks mechanism.

And, as before, we have `l:execute-strict`, which forces the operator (which may be lazy) to produce its value:

```
(define (l:execute-strict executor env continue)
  (executor env
    (extend-continuation
      (lambda (value)
        (l:advance value continue)))))
```

Its continuation must be extended as well.

But most expression handlers do not make new continuations, so there are no extensions required:

```
(define (analyze-self-evaluating expression)
  (lambda (environment continue)
    (continue expression)))

(define (analyze-variable expression)
  (lambda (environment continue)
    (continue
     (lookup-variable-value expression environment))))

(define (analyze-quoted expression)
  (let ((qval (text-of-quotation expression)))
    (lambda (environment continue)
      (continue qval))))
```

Of course, each of these handlers must be installed as the handler for the appropriate kind of expression:

```
(define-generic-procedure-handler l:analyze
  (match-args self-evaluating?)
  analyze-self-evaluating)

(define-generic-procedure-handler l:analyze
  (match-args variable?)
  analyze-variable)

(define-generic-procedure-handler l:analyze
  (match-args quoted?)
  analyze-quoted)
```

Note that the execution procedures produced by these handlers for simple expressions do not make new continutions. They just use the continuation passed to them, which is a continuation extended by the ultimate creator of that continuation.

The execution procedure produced by the analysis of a `lambda` expression is also simple. It just produces an appropriate compound procedure and passes that procedure to the continuation passed to it:

```
(define (analyze-lambda expression)
  (let ((vars (lambda-parameters expression))
        (body-exec (analyze (lambda-body expression))))
    (if (simple-parameter-list? vars)
        (lambda (environment continue)
          (continue
           (make-simple-compound-procedure vars
                                           body-exec
                                           environment)))
        (lambda (environment continue)
          (continue
           (make-complex-compound-procedure vars
                                            body-exec
                                            environment))))))
```

And, as usual, we need to declare this handler:

```
(define-generic-procedure-handler l:analyze
  (match-args lambda?)
  analyze-lambda)
```

We will not show any more of these, unless they are not obvious.

6.5.2 Handling `if`

The first real use of the continuation extensions appears in the execution procedure produced by the handler for `if`, the basic conditional.

The `analyze-if` handler is extended with a hook `l:if-hook` to allow the interpolation of work between the evaluation of the predicate part of the expression and the choice of consequent or alternative to be evauated.

This special mechanism is needed to satisfy requirement to allow the layers of the predicate value to contribute to the layers of the value of the consequent or the alternative. For example, provenance of the predicate value must be combined with the provenance of the value of the consequent or the alternative to produce the provenance of the answer.

To preserve proper tail-recursion the collection of values from the layers of the predicate and the chosen consequent or alternative must be done without creating a new continuation. So we extend continuations to allow them to accumulate the values from the contributing layers without interpolating new continuations.

```
(define (analyze-if expression)
  (let ((predicate-exec
         (analyze (if-predicate expression)))
        (consequent-exec
         (analyze (if-consequent expression)))
        (alternative-exec
         (analyze (if-alternative expression))))
    (lambda (environment continue)
      (l:execute-strict predicate-exec environment
        (extend-continuation
          (lambda (p-value)
            (l:if-hook p-value continue
              (lambda (p-value* continue*)
                ((if p-value*
                     consequent-exec
                     alternative-exec)
                 environment continue*)))))))))
```

The execution procedure produced by this handler forces the value of the predicate expression, as expected. However, the value of the predicate expression `p-value` may be a layered object rather than just a bare boolean value. To accomodate this possibility we pass the predicate expression value to a generic procedure. It will extract the base value `p-value*` and use it to select the consequent or alternative. It will also produce an appropriate continuation `continue*` to combine the results of the predicate evaluation with the results of the consequent or alternative evaluation.

```
(define l:if-hook
  (simple-generic-procedure 'l:if-hook 3
    (lambda (p-value continue receiver)
      (receiver p-value continue))))
```

The `l:if-hook` is a generic procedure that allows special handlers for things like layering (or perhaps something more interesting that we have not yet thought of!). If the predicate value is not some special thing, like a layered object, the default is just to proceed to the receiver with the predicate value and the usual continuation.

Note that the change to `if` is quite subtle, but it is not necessarily expensive. Compiled code may need to deal with the `l:if-hook` only if the `p-value` is not an explicit boolean. Of course, this will put pressure on programmers to decrease the use of non-false values of predicates as true values, but this may be a good thing.

All the rest of the special forms, like `begin`, `set!`, and `define`, and macros like `let` and `cond` are uninteresting. They are not changed from the basic interpreter.

6.5.3 Handling layered procedures

The more interesting stuff is application of procedures (possibly layered procedures!). A layered procedure does the work of using the continuation of its calling expression to do the required combination of the results of its component layers. But this is not a significant change in the interpreter. A layered procedure is just a different kind of strict procedure (an `applyer`) from the point of view of the interpreter. All that is needed is to add this case to the strict procedure apply.

```
(define (l:apply-strict procedure args continue)
  (cond ((strict-primitive-procedure? procedure)
         (continue (apply-primitive-procedure procedure args)))
        ((simple-compound-procedure? procedure)
         (l:compound-apply procedure args continue))
        ((call/cc? procedure)
         (l:deliver-continuation (car args) continue))
        ((applyer? procedure)  ; Layered procedures caught here.
         ((applyer-procedure procedure)
          procedure args continue))
        (else
         (error "Bad strict procedure" procedure args)
         'to-retain-stack)))
```

In a more serious system, we would make this a simple generic procedure that does the dispatch, allowing easy addition of more kinds of strict procedures.

The `applyer` idea is itself rather general. It is an object that can be applied as a procedure, but which carries metadata that it and other code can use. In MIT/GNU Scheme we have `entity`s and `apply-hook`s for this purpose.

The application of a layered procedure is very interesting, but it will be easier to understand, after we build up an understanding of the fundamental mechanisms, and how `if` might work. We will get back to this in section 6.5.6.

6.5.4 Continuation Extensions

Let's understand this from the bottom up. Just what is an extended continuation? It is just the continuation with the addition of the continuation marks association mechanism:

```
(define (extend-continuation continuation)
  (set-cont-marks! continuation (empty-marks))
  continuation)
```

The code `set-cont-marks!` just attaches the empty marks to the continuation as metadata.

```
(define (marks-subproblem superproblem)
  (if superproblem
      (cons '() (cont-marks superproblem))
      (root-marks)))
```

Where `cont-marks` gets the marks from the superproblem continuation.

A continuation mark is just a key-value pair that is can be associated with a continuation. In implementing layers we will need two continuation-mark keys, `mark-key:dict` and `mark-key:nested-applications`. The `mark-key:dict` will be used to obtain an association from layers to values. We could have avoided that extra layer by using the marks directly, but we were trying to isolate the layer-specific material from the continuation marks. The `mark-key:nested-applications` is just a counter for the number of outstanding applications, which will become zero when the continuation is allowed to proceed.

6.5.5 Layers and `if`

Now, let's look at how `l:if-hook` actually works if the predicate value is actually a layered object:

```
(define-generic-procedure-handler l:if-hook
  (match-args
    layered-thing? extended-continuation? any-object?)
  (lambda (p-value continue receiver)
    (receiver (base-value p-value)
              (merge-predicate-layer-values p-value
                                            continue))))
```

It calls the receiver (from `analyze-if`):

```
(lambda (p-value* continue*)
  ((if p-value*
       consequent-exec
       alternative-exec)
   environment continue*))
```

with the base value of the predicate value, that the receiver can use to do the selection of consequent or alternative, and the continuation, which will be a mechanism for merging the predicate values layers with the results of executing the chosen consequent or alternative in the continuation of the `if` expression.

The procedure `merge-predicate-layer-values` merges the appropriate predicate value layers into the continuation `cont` of the `if` expression and then returns a continuation that will merge in the chosen consequent or alternative values into the same continuation. If there is no more to be done the continuation is then allowed to proceed.

```
(define (merge-predicate-layer-values p-value cont)
  (accept-values p-merge p-value cont)
  (extend-continuation
    (lambda (c/a-value)
      (accept-values c/a-merge c/a-value cont)
      (return-if-complete cont))))
```

The real work is done by `accept-values`. For every layer in the value passed in it uses a layer-specific merge procedure to combine the value for that layer stored in the continuation with the value for that layer in the value.

```
(define (accept-values get-merger value cont)
  (for-each
   (lambda (layer-name)
     (value-receiver layer-name
                     ((get-merger layer-name)
                      (((cont-dict cont) 'get-value) layer-name)
                      (get-layer-value-or-default layer-name
                                                  value))
                     cont))
   (available-layers value)))
```

Notice that there needs to be a merge procedure and a possible default value for each layer. This must be set up in the description of a layer. So, for example, for tracking provenance the merger will be a union of the support sets.

The merged result is passed to the `value-receiver`, which then stashes it, under than layer name, into the waiting continuation.

```
(define (value-receiver layer-name new-value cont)
  (cond ((ignore-value? new-value)
         'nothing-to-do)
        (else
         (((cont-dict cont) 'update!) layer-name new-value)
         'updated-layer-with-new-value)))
```

When all nested applications have completely finished with this continuation, the continuation is allowed to proceed, with a newly constructed layered object as its result:

```
(define (return-if-complete cont)
  (if (= (cont-nested-apps cont) 0)
      (cont
       (make-layered-thing (((cont-dict cont) 'filled-entries))))
      'multiple-nested-applications))
```

So this is how `if` is extended for predicate values that are layered.

6.5.6 Application of layered procedures

When an object of type `applyer` is the procedure sent to `l:apply-strict` the `applyer-procedure` of the object is extracted. For a layered procedure the applyer procedure is `layered-procedure-dispatch`:

```
(define (layered-procedure-dispatch procedure args continue)
  (layers-processor procedure
                    args
                    (apply lset-union eq?
                           (available-layers procedure)
                           (map available-layers args))
                    (process-layers continue))
  (end-of-layers continue))
```

This procedure has to coordinate the processing of each of the layers that are present in the arguments or the procedure. The `process-layers` procedure returns a processor for each layer, such that the results are targeted to the continuation of the calling expression of the layered procedure. It starts by incrementing the nested applications counter for the continuation. This will be decremented by the `end-of-layers` procedure after all the lay-

ers are processed. The procedure `process-layers` returns the `process-a-layer` to be used for each layer:

```
(define (process-layers cont)
  (cont-increment-nested-apps cont)
  (define (process-a-layer layer-name updater)
    (updater (((cont-dict cont) 'get-value) layer-name)
             (extend-continuation
               (lambda (new-value)
                 (if (and (eq? layer-name 'base)
                          (layered-thing? new-value))
                     (accept-values apply-merge new-value cont)
                     (value-receiver layer-name new-value
                                     cont))))))
  process-a-layer)
```

The `process-a-layer` procedure grabs the current value stored in the continuation with the given layer name and passes that value with a procedure that will take the new value from running the procedure layer handler and stash it in the continuation, to the updater that will actually run the procedure layer. This is a pretty baroque handshake, so it probably can be simplified.

The `layers-processor` procedure iterates through the layers, processing each layer with the target continuation given to `process-layers`.

```
(define (layers-processor procedure args relevant-layers
                          process-a-layer)
  (for-each (lambda (layer-name)
              (process-a-layer layer-name
                (layer-applicator procedure args layer-name)))
            relevant-layers))
```

The layer applicator is actually where the procedure layer code is executed. The base layer is treated separately, because the other layer handlers may need the current value from the continuation to merge with the new information. This value is passed in as an extra first argument to the handler for the layer.

```
(define (layer-applicator procedure args layer-name)

  (define (base-layer-applicator procedure args continue)
    (let ((base-proc (base-value procedure)))
      (l:apply-strict base-proc
                      (if (strict-primitive-procedure? base-proc)
                          (map base-value args)
                          args)
                      continue)))

  (lambda (cont-layer-value continue)
    (cond ((eq? layer-name 'base)
           (base-layer-applicator procedure
                                  args
                                  continue))
          ((any (lambda (arg)      ; A changeable policy.
                  (memq layer-name
                        (available-layers arg)))
                args)
           (l:apply-strict (get-layer-value-or-default
                            layer-name procedure)
                           (cons cont-layer-value args)
                           continue))
          (else 'OK))))
```

Notice that both for application and for `if` we keep track of the number of nested applications that have to be finished for the continuation of the expression to proceed.

```
(define (end-of-layers cont)
  (cont-decrement-nested-apps cont)
  (return-if-complete cont))
```

This is still a bit mysterious.

6.6 The promise of layering

We have only scratched the surface of what can be done with an easy and convenient mechanism for layering of data and programs. It is an open area of research. The development of systems to support such layering can have huge consequence for the future.

Sensitivity analysis is an important feature that can be built using annotated data and layered procedures. For example, in mechanics, if we have a system that evolves the solution of a system of differential equations from some initial conditions, it is often valu-

able to understand the way a tube of trajectories that surround a reference trajectory deforms. This is usually accomplished by integrating a variational system along with the reference trajectory. Similarly, it may be possible to carry a probability distribution of values around a nominal value along with the nominal value computed in some analyses. This may be accomplished by annotating the values with distributions and providing the operations with overlaying procedures to combine the distributions, guided by the nominals, perhaps implementing Bayesian analysis. Of course, to do this well is not easy.

An even more exciting but related idea is that of perturbational programming. By analogy with the differential equations example, can we program symbolic systems to carry a "tube" of variations around a reference trajectory, thus allowing us to consider small variations of a query? Consider, for example, the problem of doing a search. Given a set of keywords, the system does some magic that comes up with a list of documents that match the keywords. Suppose we incrementally change a single keyword. How sensitive is the search to that keyword? More important, is it possible to reuse some of the work that was done getting the previous result in the incrementally different search? We don't know the answers to these questions, but if it is possible, we want to be able to capture the methods by a kind of perturbational program, built as an overlay on the base program.

Dependencies mitigate inconsistency

Dependency annotations on data give us a powerful tool for organizing human-like computations. For example, all humans harbor mutually inconsistent beliefs: an intelligent person may be committed to the scientific method yet have a strong attachment to some superstitious or ritual practices; a person may have a strong belief in the sanctity of all human life, yet also believe that capital punishment is sometimes justified. If we were really logicians this kind of inconsistency would be fatal: if we really were to simultaneously believe both propositions P and NOT P, then we would have to believe all propositions! But somehow we manage to keep inconsistent beliefs from inhibiting all useful thought. Our personal belief systems appear to be locally consistent, in that there are no contradictions apparent. If we observe inconsistencies we do not crash; we may feel conflicted or we may chuckle.

We can attach to each proposition a set of supporting assumptions, allowing deductions to be conditional on the assumption set. Then, if a contradiction occurs, a process can determine the particular "nogood set" of inconsistent assumptions. The system can then "chuckle," realizing that no deductions based on any superset of those assumptions can be believed. This chuckling process, dependency-directed backtracking, can be used to optimize a complex search process, allowing a search to make the best use of its mistakes. But enabling a process to simultaneously hold beliefs based on mutually inconsistent sets of assumptions without logical disaster is revolutionary.

Restrictions on the use of data

Data is often encumbered by restrictions on the ways it may be used. These encumberances may be determined by statute, by contract, by custom, or by common decency. Some of these restrictions are intended to control the diffusion of the data, while others are intended to delimit the consequences of actions predicated on that data.

The allowable uses of data may be further restricted by the sender: "I am telling you this information in confidence. You may not use it to compete with me, and you may not give it to any of my competitors." Data may also be restricted by the receiver: "I don't want to know anything about this that I may not tell my spouse."

Although the details may be quite involved, as data is passed from one individual or organization to another, the restrictions on the uses to which it may be put are changed in ways that can often be formulated as algebraic expressions. These expressions describe how the restrictions on the use of a particular data item may be computed from the history of its transmission: the encumberances that are added or deleted at each step. When parts of one data set are combined with parts of another data set, the restrictions on the ways that the extracts may be used and the restrictions on the ways that they may be combined must determine the restrictions on the combination. A formalization of this process is a *data-purpose algebra* [54] description.

Data-purpose algebra layers can be helpful in building systems that track the distribution and use of sensitive data to enable auditing and to inhibit the misuse of that data. But this kind of application is much larger than just a simple matter of layering.

To make it effective requires ways of ensuring the security of the process, to prevent leakage through uncontrolled channels or compromise of the tracking layers. There is a great deal of research to be done here.