4

Metalinguistic Abstraction

> . . . It's in words that the magic is—Abracadabra, Open Sesame, and the rest—but the magic words in one story aren't magical in the next. The real magic is to understand which words work, and when, and for what; the trick is to learn the trick.
>
> . . . And those words are made from the letters of our alphabet: a couple-dozen squiggles we can draw with the pen. This is the key! And the treasure, too, if we can only get our hands on it! It's as if—as if the key to the treasure *is* the treasure!
>
> —John Barth, *Chimera*

In our study of program design, we have seen that expert programmers control the complexity of their designs with the same general techniques used by designers of all complex systems. They combine primitive elements to form compound objects, they abstract compound

objects to form higher-level building blocks, and they preserve modularity by adopting appropriate large-scale views of system structure. In illustrating these techniques, we have used Lisp as a language for describing processes and for constructing computational data objects and processes to model complex phenomena in the real world. However, as we confront increasingly complex problems, we will find that Lisp, or indeed any fixed programming language, is not sufficient for our needs. We must constantly turn to new languages in order to express our ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe (and hence to think about) the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand.[1]

Programming is endowed with a multitude of languages. There are

[1]The same idea is pervasive throughout all of engineering. For example, electrical engineers use many different languages for describing circuits. Two of these are the language of electrical *networks* and the language of electrical *systems*. The network language emphasizes the physical modeling of devices in terms of discrete electrical elements. The primitive objects of the network language are primitive electrical components such as resistors, capacitors, inductors, and transistors, which are characterized in terms of physical variables called voltage and current. When describing circuits in the network language, the engineer is concerned with the physical characteristics of a design. In contrast, the primitive objects of the system language are signal-processing modules such as filters and amplifiers. Only the functional behavior of the modules is relevant, and signals are manipulated without concern for their physical realization as voltages and currents. The system language is erected on the network language, in the sense that the elements of signal-processing systems are constructed from electrical networks. Here, however, the concerns are with the large-scale organization of electrical devices to solve a given application problem; the physical feasibility of the parts is assumed. This layered collection of languages is another example of the stratified design technique illustrated by the picture language of Section 2.2.4.

physical languages, such as the machine languages for particular computers. These languages are concerned with the representation of data and control in terms of individual bits of storage and primitive machine instructions. The machine-language programmer is concerned with using the given hardware to erect systems and utilities for the efficient implementation of resource-limited computations. High-level languages, erected on a machine-language substrate, hide concerns about the representation of data as collections of bits and the representation of programs as sequences of primitive instructions. These languages have means of combination and abstraction, such as procedure definition, that are appropriate to the larger-scale organization of systems.

Metalinguistic abstraction—establishing new languages—plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators. An *evaluator* (or *interpreter*) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.

It is no exaggeration to regard this as the most fundamental idea in programming:

> The evaluator, which determines the meaning of expressions in a programming language, is just another program.

To appreciate this point is to change our images of ourselves as programmers. We come to see ourselves as designers of languages, rather than only users of languages designed by others.

In fact, we can regard almost any program as the evaluator for some language. For instance, the polynomial manipulation system of Section

2.5.3 embodies the rules of polynomial arithmetic and implements them in terms of operations on list-structured data. If we augment this system with procedures to read and print polynomial expressions, we have the core of a special-purpose language for dealing with problems in symbolic mathematics. The digital-logic simulator of Section 3.3.4 and the constraint propagator of Section 3.3.5 are legitimate languages in their own right, each with its own primitives, means of combination, and means of abstraction. Seen from this perspective, the technology for coping with large-scale computer systems merges with the technology for building new computer languages, and computer science itself becomes no more (and no less) than the discipline of constructing appropriate descriptive languages.

We now embark on a tour of the technology by which languages are established in terms of other languages. In this chapter we shall use Lisp as a base, implementing evaluators as Lisp procedures. Lisp is particularly well suited to this task, because of its ability to represent and manipulate symbolic expressions. We will take the first step in understanding how languages are implemented by building an evaluator for Lisp itself. The language implemented by our evaluator will be a subset of the Scheme dialect of Lisp that we use in this book. Although the evaluator described in this chapter is written for a particular dialect of Lisp, it contains the essential structure of an evaluator for any expression-oriented language designed for writing programs for a sequential machine. (In fact, most language processors contain, deep within them, a little "Lisp" evaluator.) The evaluator has been simplified for the purposes of illustration and discussion, and some features have been left out that would be important to include in a production-quality Lisp system. Nevertheless, this simple evaluator is adequate to execute most of the programs

in this book.[2]

An important advantage of making the evaluator accessible as a Lisp program is that we can implement alternative evaluation rules by describing these as modifications to the evaluator program. One place where we can use this power to good effect is to gain extra control over the ways in which computational models embody the notion of time, which was so central to the discussion in Chapter 3. There, we mitigated some of the complexities of state and assignment by using streams to decouple the representation of time in the world from time in the computer. Our stream programs, however, were sometimes cumbersome, because they were constrained by the applicative-order evaluation of Scheme. In Section 4.2, we'll change the underlying language to provide for a more elegant approach, by modifying the evaluator to provide for *normal-order evaluation*.

Section 4.3 implements a more ambitious linguistic change, whereby expressions have many values, rather than just a single value. In this language of *nondeterministic computing*, it is natural to express processes that generate all possible values for expressions and then search for those values that satisfy certain constraints. In terms of models of computation and time, this is like having time branch into a set of "possible futures" and then searching for appropriate time lines. With our nondeterministic evaluator, keeping track of multiple values and performing searches are handled automatically by the underlying mechanism of the language.

In Section 4.4 we implement a *logic-programming* language in which

[2] The most important features that our evaluator leaves out are mechanisms for handling errors and supporting debugging. For a more extensive discussion of evaluators, see Friedman et al. 1992, which gives an exposition of programming languages that proceeds via a sequence of evaluators written in Scheme.

knowledge is expressed in terms of relations, rather than in terms of computations with inputs and outputs. Even though this makes the language drastically different from Lisp, or indeed from any conventional language, we will see that the logic-programming evaluator shares the essential structure of the Lisp evaluator.

4.1 The Metacircular Evaluator

Our evaluator for Lisp will be implemented as a Lisp program. It may seem circular to think about evaluating Lisp programs using an evaluator that is itself implemented in Lisp. However, evaluation is a process, so it is appropriate to describe the evaluation process using Lisp, which, after all, is our tool for describing processes.[3] An evaluator that is written in the same language that it evaluates is said to be *metacircular*.

The metacircular evaluator is essentially a Scheme formulation of the environment model of evaluation described in Section 3.2. Recall that the model has two basic parts:

1. To evaluate a combination (a compound expression other than a special form), evaluate the subexpressions and then apply the value of the operator subexpression to the values of the operand subexpressions.

2. To apply a compound procedure to a set of arguments, evaluate the body of the procedure in a new environment. To construct

[3] Even so, there will remain important aspects of the evaluation process that are not elucidated by our evaluator. The most important of these are the detailed mechanisms by which procedures call other procedures and return values to their callers. We will address these issues in Chapter 5, where we take a closer look at the evaluation process by implementing the evaluator as a simple register machine.

this environment, extend the environment part of the procedure object by a frame in which the formal parameters of the procedure are bound to the arguments to which the procedure is applied.

These two rules describe the essence of the evaluation process, a basic cycle in which expressions to be evaluated in environments are reduced to procedures to be applied to arguments, which in turn are reduced to new expressions to be evaluated in new environments, and so on, until we get down to symbols, whose values are looked up in the environment, and to primitive procedures, which are applied directly (see Figure 4.1).[4] This evaluation cycle will be embodied by the interplay between the two critical procedures in the evaluator, `eval` and `apply`, which are described in Section 4.1.1 (see Figure 4.1).

The implementation of the evaluator will depend upon procedures

[4]If we grant ourselves the ability to apply primitives, then what remains for us to implement in the evaluator? The job of the evaluator is not to specify the primitives of the language, but rather to provide the connective tissue—the means of combination and the means of abstraction—that binds a collection of primitives to form a language. Specifically:

• The evaluator enables us to deal with nested expressions. For example, although simply applying primitives would suffice for evaluating the expression `(+ 1 6)`, it is not adequate for handling `(+ 1 (* 2 3))`. As far as the primitive procedure `+` is concerned, its arguments must be numbers, and it would choke if we passed it the expression `(* 2 3)` as an argument. One important role of the evaluator is to choreograph procedure composition so that `(* 2 3)` is reduced to 6 before being passed as an argument to `+`.

• The evaluator allows us to use variables. For example, the primitive procedure for addition has no way to deal with expressions such as `(+ x 1)`. We need an evaluator to keep track of variables and obtain their values before invoking the primitive procedures.

• The evaluator allows us to define compound procedures. This involves keeping track of procedure definitions, knowing how to use these definitions in evaluating expressions, and providing a mechanism that enables procedures to accept arguments.

• The evaluator provides the special forms, which must be evaluated differently from procedure calls.

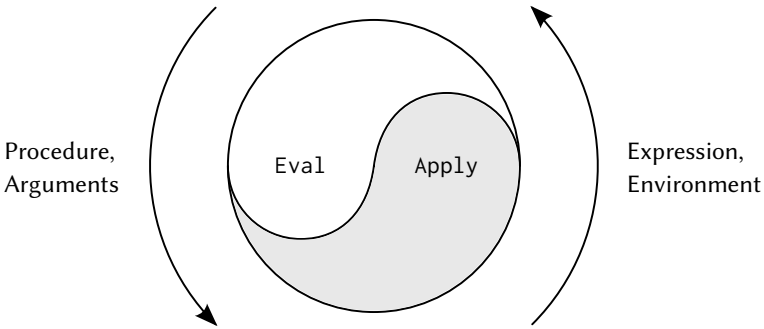

Figure 4.1: The `eval-apply` cycle exposes the essence of a computer language.

that define the *syntax* of the expressions to be evaluated. We will use data abstraction to make the evaluator independent of the representation of the language. For example, rather than committing to a choice that an assignment is to be represented by a list beginning with the symbol `set!` we use an abstract predicate `assignment?` to test for an assignment, and we use abstract selectors `assignment-variable` and `assignment-value` to access the parts of an assignment. Implementation of expressions will be described in detail in Section 4.1.2. There are also operations, described in Section 4.1.3, that specify the representation of procedures and environments. For example, `make-procedure` constructs compound procedures, `lookup-variable-value` accesses the values of variables, and `apply-primitive-procedure` applies a primitive procedure to a given list of arguments.

4.1.1 The Core of the Evaluator

The evaluation process can be described as the interplay between two procedures: `eval` and `apply`.

Eval

`eval` takes as arguments an expression and an environment. It classifies the expression and directs its evaluation. `eval` is structured as a case analysis of the syntactic type of the expression to be evaluated. In order to keep the procedure general, we express the determination of the type of an expression abstractly, making no commitment to any particular representation for the various types of expressions. Each type of expression has a predicate that tests for it and an abstract means for selecting its parts. This *abstract syntax* makes it easy to see how we can change the syntax of the language by using the same evaluator, but with a different collection of syntax procedures.

Primitive expressions

- For self-evaluating expressions, such as numbers, `eval` returns the expression itself.
- `eval` must look up variables in the environment to find their values.

Special forms

- For quoted expressions, `eval` returns the expression that was quoted.
- An assignment to (or a definition of) a variable must recursively call `eval` to compute the new value to be associated with the variable. The environment must be modified to change (or create) the binding of the variable.

- An `if` expression requires special processing of its parts, so as to evaluate the consequent if the predicate is true, and otherwise to evaluate the alternative.
- A `lambda` expression must be transformed into an applicable procedure by packaging together the parameters and body specified by the `lambda` expression with the environment of the evaluation.
- A `begin` expression requires evaluating its sequence of expressions in the order in which they appear.
- A case analysis (`cond`) is transformed into a nest of `if` expressions and then evaluated.

Combinations

- For a procedure application, `eval` must recursively evaluate the operator part and the operands of the combination. The resulting procedure and arguments are passed to `apply`, which handles the actual procedure application.

Here is the definition of `eval`:

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp) (make-procedure (lambda-parameters exp)
                                       (lambda-body exp)
                                       env))
```

```
((begin? exp)
 (eval-sequence (begin-actions exp) env))
((cond? exp) (eval (cond->if exp) env))
((application? exp)
 (apply (eval (operator exp) env)
        (list-of-values (operands exp) env)))
(else
 (error "Unknown expression type: EVAL" exp))))
```

For clarity, `eval` has been implemented as a case analysis using `cond`. The disadvantage of this is that our procedure handles only a few distinguishable types of expressions, and no new ones can be defined without editing the definition of `eval`. In most Lisp implementations, dispatching on the type of an expression is done in a data-directed style. This allows a user to add new types of expressions that `eval` can distinguish, without modifying the definition of `eval` itself. (See Exercise 4.3.)

Apply

`apply` takes two arguments, a procedure and a list of arguments to which the procedure should be applied. `apply` classifies procedures into two kinds: It calls `apply-primitive-procedure` to apply primitives; it applies compound procedures by sequentially evaluating the expressions that make up the body of the procedure. The environment for the evaluation of the body of a compound procedure is constructed by extending the base environment carried by the procedure to include a frame that binds the parameters of the procedure to the arguments to which the procedure is to be applied. Here is the definition of `apply`:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
```

```
((compound-procedure? procedure)
 (eval-sequence
   (procedure-body procedure)
   (extend-environment
     (procedure-parameters procedure)
     arguments
     (procedure-environment procedure))))
(else
 (error
  "Unknown procedure type: APPLY" procedure))))
```

Procedure arguments

When `eval` processes a procedure application, it uses `list-of-values` to produce the list of arguments to which the procedure is to be applied. `list-of-values` takes as an argument the operands of the combination. It evaluates each operand and returns a list of the corresponding values:[5]

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

[5] We could have simplified the `application?` clause in `eval` by using `map` (and stipulating that `operands` returns a list) rather than writing an explicit `list-of-values` procedure. We chose not to use `map` here to emphasize the fact that the evaluator can be implemented without any use of higher-order procedures (and thus could be written in a language that doesn't have higher-order procedures), even though the language that it supports will include higher-order procedures.

Conditionals

`eval-if` evaluates the predicate part of an `if` expression in the given environment. If the result is true, `eval-if` evaluates the consequent, otherwise it evaluates the alternative:

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

The use of `true?` in `eval-if` highlights the issue of the connection between an implemented language and an implementation language. The `if-predicate` is evaluated in the language being implemented and thus yields a value in that language. The interpreter predicate `true?` translates that value into a value that can be tested by the `if` in the implementation language: The metacircular representation of truth might not be the same as that of the underlying Scheme.[6]

Sequences

`eval-sequence` is used by `apply` to evaluate the sequence of expressions in a procedure body and by `eval` to evaluate the sequence of expressions in a `begin` expression. It takes as arguments a sequence of expressions and an environment, and evaluates the expressions in the order in which they occur. The value returned is the value of the final expression.

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps)
         (eval (first-exp exps) env))
```

[6] In this case, the language being implemented and the implementation language are the same. Contemplation of the meaning of `true?` here yields expansion of consciousness without the abuse of substance.

```
        (else
         (eval (first-exp exps) env)
         (eval-sequence (rest-exps exps) env))))
```

Assignments and definitions

The following procedure handles assignments to variables. It calls `eval` to find the value to be assigned and transmits the variable and the resulting value to `set-variable-value!` to be installed in the designated environment.

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
```

Definitions of variables are handled in a similar manner.[7]

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

We have chosen here to return the symbol `ok` as the value of an assignment or a definition.[8]

> **Exercise 4.1:** Notice that we cannot tell whether the metacircular evaluator evaluates operands from left to right or from

[7] This implementation of `define` ignores a subtle issue in the handling of internal definitions, although it works correctly in most cases. We will see what the problem is and how to solve it in Section 4.1.6.

[8] As we said when we introduced `define` and `set!`, these values are implementation-dependent in Scheme—that is, the implementor can choose what value to return.

right to left. Its evaluation order is inherited from the underlying Lisp: If the arguments to `cons` in `list-of-values` are evaluated from left to right, then `list-of-values` will evaluate operands from left to right; and if the arguments to `cons` are evaluated from right to left, then `list-of-values` will evaluate operands from right to left.

Write a version of `list-of-values` that evaluates operands from left to right regardless of the order of evaluation in the underlying Lisp. Also write a version of `list-of-values` that evaluates operands from right to left.

4.1.2 Representing Expressions

The evaluator is reminiscent of the symbolic differentiation program discussed in Section 2.3.2. Both programs operate on symbolic expressions. In both programs, the result of operating on a compound expression is determined by operating recursively on the pieces of the expression and combining the results in a way that depends on the type of the expression. In both programs we used data abstraction to decouple the general rules of operation from the details of how expressions are represented. In the differentiation program this meant that the same differentiation procedure could deal with algebraic expressions in prefix form, in infix form, or in some other form. For the evaluator, this means that the syntax of the language being evaluated is determined solely by the procedures that classify and extract pieces of expressions.

Here is the specification of the syntax of our language:

- The only self-evaluating items are numbers and strings:

```
(define (self-evaluating? exp)
  (cond ((number? exp) true)
```

```
        ((string? exp) true)
        (else false)))
```

- Variables are represented by symbols:

```
(define (variable? exp) (symbol? exp))
```

- Quotations have the form `(quote` ⟨*text-of-quotation*⟩`)`:[9]

```
(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))
```

  `quoted?` is defined in terms of the procedure `tagged-list?`, which identifies lists beginning with a designated symbol:

```
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

- Assignments have the form `(set!` ⟨*var*⟩ ⟨*value*⟩`)`:

```
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))
```

- Definitions have the form

```
(define ⟨var⟩ ⟨value⟩)
```

  or the form

[9] As mentioned in Section 2.3.1, the evaluator sees a quoted expression as a list beginning with `quote`, even if the expression is typed with the quotation mark. For example, the expression `'a` would be seen by the evaluator as `(quote a)`. See Exercise 2.55.

```
(define (⟨var⟩ ⟨parameter_1⟩ ... ⟨parameter_n⟩)
  ⟨body⟩)
```

The latter form (standard procedure definition) is syntactic sugar for

```
(define ⟨var⟩
  (lambda (⟨parameter_1⟩ ... ⟨parameter_n⟩)
    ⟨body⟩))
```

The corresponding syntax procedures are the following:

```
(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)      ; formal parameters
                   (cddr exp))))    ; body
```

- `lambda` expressions are lists that begin with the symbol `lambda`:

```
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))
```

We also provide a constructor for `lambda` expressions, which is used by `definition-value`, above:

```
(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

- Conditionals begin with `if` and have a predicate, a consequent, and an (optional) alternative. If the expression has no alternative part, we provide `false` as the alternative.[10]

```
(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
```

  We also provide a constructor for `if` expressions, to be used by `cond->if` to transform `cond` expressions into `if` expressions:

```
(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

- `begin` packages a sequence of expressions into a single expression. We include syntax operations on `begin` expressions to extract the actual sequence from the `begin` expression, as well as selectors that return the first expression and the rest of the expressions in the sequence.[11]

```
(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions exp) (cdr exp))
```

[10] The value of an `if` expression when the predicate is false and there is no alternative is unspecified in Scheme; we have chosen here to make it false. We will support the use of the variables `true` and `false` in expressions to be evaluated by binding them in the global environment. See Section 4.1.4.

[11] These selectors for a list of expressions—and the corresponding ones for a list of operands—are not intended as a data abstraction. They are introduced as mnemonic names for the basic list operations in order to make it easier to understand the explicit-control evaluator in Section 5.4.

```
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
```

We also include a constructor `sequence->exp` (for use by `cond->if`) that transforms a sequence into a single expression, using `begin` if necessary:

```
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin seq) (cons 'begin seq))
```

- A procedure application is any compound expression that is not one of the above expression types. The `car` of the expression is the operator, and the `cdr` is the list of operands:

```
(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

Derived expressions

Some special forms in our language can be defined in terms of expressions involving other special forms, rather than being implemented directly. One example is `cond`, which can be implemented as a nest of `if` expressions. For example, we can reduce the problem of evaluating the expression

```
(cond ((> x 0) x)
      ((= x 0) (display 'zero) 0)
      (else (- x)))
```

to the problem of evaluating the following expression involving `if` and `begin` expressions:

```
(if (> x 0)
    x
    (if (= x 0)
        (begin (display 'zero) 0)
        (- x)))
```

Implementing the evaluation of `cond` in this way simplifies the evaluator because it reduces the number of special forms for which the evaluation process must be explicitly specified.

We include syntax procedures that extract the parts of a `cond` expression, and a procedure `cond->if` that transforms `cond` expressions into `if` expressions. A case analysis begins with `cond` and has a list of predicate-action clauses. A clause is an `else` clause if its predicate is the symbol `else`.[12]

```
(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond->if exp) (expand-clauses (cond-clauses exp)))
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
```

[12]The value of a `cond` expression when all the predicates are false and there is no `else` clause is unspecified in Scheme; we have chosen here to make it false.

```
(let ((first (car clauses))
      (rest (cdr clauses)))
  (if (cond-else-clause? first)
      (if (null? rest)
          (sequence->exp (cond-actions first))
          (error "ELSE clause isn't last: COND->IF"
                 clauses))
      (make-if (cond-predicate first)
               (sequence->exp (cond-actions first))
               (expand-clauses rest))))))
```

Expressions (such as cond) that we choose to implement as syntactic transformations are called *derived expressions*. `let` expressions are also derived expressions (see Exercise 4.6).[13]

> **Exercise 4.2:** Louis Reasoner plans to reorder the cond clauses in `eval` so that the clause for procedure applications appears before the clause for assignments. He argues that this will make the interpreter more efficient: Since programs usually contain more applications than assignments, definitions, and so on, his modified `eval` will usually check fewer clauses than the original `eval` before identifying the type of an expression.
>
> a. What is wrong with Louis's plan? (Hint: What will

[13] Practical Lisp systems provide a mechanism that allows a user to add new derived expressions and specify their implementation as syntactic transformations without modifying the evaluator. Such a user-defined transformation is called a *macro*. Although it is easy to add an elementary mechanism for defining macros, the resulting language has subtle name-conflict problems. There has been much research on mechanisms for macro definition that do not cause these difficulties. See, for example, Kohlbecker 1986, Clinger and Rees 1991, and Hanson 1991.

Louis's evaluator do with the expression `(define x 3)`?)

b. Louis is upset that his plan didn't work. He is willing to go to any lengths to make his evaluator recognize procedure applications before it checks for most other kinds of expressions. Help him by changing the syntax of the evaluated language so that procedure applications start with `call`. For example, instead of `(factorial 3)` we will now have to write `(call factorial 3)` and instead of `(+ 1 2)` we will have to write `(call + 1 2)`.

Exercise 4.3: Rewrite `eval` so that the dispatch is done in data-directed style. Compare this with the data-directed differentiation procedure of Exercise 2.73. (You may use the `car` of a compound expression as the type of the expression, as is appropriate for the syntax implemented in this section.)

Exercise 4.4: Recall the definitions of the special forms `and` and `or` from Chapter 1:

- `and`: The expressions are evaluated from left to right. If any expression evaluates to false, false is returned; any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then true is returned.
- `or`: The expressions are evaluated from left to right. If any expression evaluates to a true value, that value

is returned; any remaining expressions are not evaluated. If all expressions evaluate to false, or if there are no expressions, then false is returned.

Install `and` and `or` as new special forms for the evaluator by defining appropriate syntax procedures and evaluation procedures `eval-and` and `eval-or`. Alternatively, show how to implement `and` and `or` as derived expressions.

Exercise 4.5: Scheme allows an additional syntax for `cond` clauses, (⟨*test*⟩ `=>` ⟨*recipient*⟩). If ⟨*test*⟩ evaluates to a true value, then ⟨*recipient*⟩ is evaluated. Its value must be a procedure of one argument; this procedure is then invoked on the value of the ⟨*test*⟩, and the result is returned as the value of the `cond` expression. For example

```
(cond ((assoc 'b '((a 1) (b 2))) => cadr)
      (else false))
```

returns 2. Modify the handling of `cond` so that it supports this extended syntax.

Exercise 4.6: `let` expressions are derived expressions, because

```
(let ((⟨var1⟩ ⟨exp1⟩) ... (⟨varn⟩ ⟨expn⟩))
  ⟨body⟩)
```

is equivalent to

```
((lambda (⟨var1⟩ ... ⟨varn⟩)
   ⟨body⟩)
 ⟨exp1⟩
 ...
 ⟨expn⟩)
```

Implement a syntactic transformation `let->combination` that reduces evaluating `let` expressions to evaluating combinations of the type shown above, and add the appropriate clause to `eval` to handle `let` expressions.

Exercise 4.7: `let*` is similar to `let`, except that the bindings of the `let*` variables are performed sequentially from left to right, and each binding is made in an environment in which all of the preceding bindings are visible. For example

```
(let* ((x 3)  (y (+ x 2))  (z (+ x y 5)))
  (* x z))
```

returns 39. Explain how a `let*` expression can be rewritten as a set of nested `let` expressions, and write a procedure `let*->nested-lets` that performs this transformation. If we have already implemented `let` (Exercise 4.6) and we want to extend the evaluator to handle `let*`, is it sufficient to add a clause to `eval` whose action is

```
(eval (let*->nested-lets exp) env)
```

or must we explicitly expand `let*` in terms of non-derived expressions?

Exercise 4.8: "Named `let`" is a variant of `let` that has the form

```
(let ⟨var⟩ ⟨bindings⟩ ⟨body⟩)
```

The ⟨*bindings*⟩ and ⟨*body*⟩ are just as in ordinary `let`, except that ⟨*var*⟩ is bound within ⟨*body*⟩ to a procedure whose body is ⟨*body*⟩ and whose parameters are the variables in

the ⟨*bindings*⟩. Thus, one can repeatedly execute the ⟨*body*⟩ by invoking the procedure named ⟨*var*⟩. For example, the iterative Fibonacci procedure (Section 1.2.2) can be rewritten using named `let` as follows:

```
(define (fib n)
  (let fib-iter ((a 1)
                 (b 0)
                 (count n))
    (if (= count 0)
        b
        (fib-iter (+ a b) a (- count 1)))))
```

Modify `let->combination` of Exercise 4.6 to also support named `let`.

Exercise 4.9: Many languages support a variety of iteration constructs, such as `do`, `for`, `while`, and `until`. In Scheme, iterative processes can be expressed in terms of ordinary procedure calls, so special iteration constructs provide no essential gain in computational power. On the other hand, such constructs are often convenient. Design some iteration constructs, give examples of their use, and show how to implement them as derived expressions.

Exercise 4.10: By using data abstraction, we were able to write an `eval` procedure that is independent of the particular syntax of the language to be evaluated. To illustrate this, design and implement a new syntax for Scheme by modifying the procedures in this section, without changing `eval` or `apply`.

4.1.3 Evaluator Data Structures

In addition to defining the external syntax of expressions, the evaluator implementation must also define the data structures that the evaluator manipulates internally, as part of the execution of a program, such as the representation of procedures and environments and the representation of true and false.

Testing of predicates

For conditionals, we accept anything to be true that is not the explicit `false` object.

```
(define (true? x)  (not (eq? x false)))
(define (false? x) (eq? x false))
```

Representing procedures

To handle primitives, we assume that we have available the following procedures:

- `(apply-primitive-procedure` ⟨*proc*⟩ ⟨*args*⟩`)`

  applies the given primitive procedure to the argument values in the list ⟨*args*⟩ and returns the result of the application.

- `(primitive-procedure?` ⟨*proc*⟩`)`

  tests whether ⟨*proc*⟩ is a primitive procedure.

These mechanisms for handling primitives are further described in Section 4.1.4.

Compound procedures are constructed from parameters, procedure bodies, and environments using the constructor `make-procedure`:

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? p)
  (tagged-list? p 'procedure))
(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

Operations on Environments

The evaluator needs operations for manipulating environments. As explained in Section 3.2, an environment is a sequence of frames, where each frame is a table of bindings that associate variables with their corresponding values. We use the following operations for manipulating environments:

- `(lookup-variable-value` ⟨*var*⟩ ⟨*env*⟩`)`

  returns the value that is bound to the symbol ⟨*var*⟩ in the environment ⟨*env*⟩, or signals an error if the variable is unbound.

- `(extend-environment` ⟨*variables*⟩ ⟨*values*⟩ ⟨*base-env*⟩`)`

  returns a new environment, consisting of a new frame in which the symbols in the list ⟨*variables*⟩ are bound to the corresponding elements in the list ⟨*values*⟩, where the enclosing environment is the environment ⟨*base-env*⟩.

- `(define-variable!` ⟨*var*⟩ ⟨*value*⟩ ⟨*env*⟩`)`

  adds to the first frame in the environment ⟨*env*⟩ a new binding that associates the variable ⟨*var*⟩ with the value ⟨*value*⟩.

- `(set-variable-value!` ⟨*var*⟩ ⟨*value*⟩ ⟨*env*⟩`)`

  changes the binding of the variable ⟨*var*⟩ in the environment ⟨*env*⟩ so that the variable is now bound to the value ⟨*value*⟩, or signals an error if the variable is unbound.

To implement these operations we represent an environment as a list of frames. The enclosing environment of an environment is the `cdr` of the list. The empty environment is simply the empty list.

```
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())
```

Each frame of an environment is represented as a pair of lists: a list of the variables bound in that frame and a list of the associated values.[14]

```
(define (make-frame variables values)
  (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

To extend an environment by a new frame that associates variables with values, we make a frame consisting of the list of variables and the list of values, and we adjoin this to the environment. We signal an error if the number of variables does not match the number of values.

[14] Frames are not really a data abstraction in the following code: `set-variable-value!` and `define-variable!` use `set-car!` to directly modify the values in a frame. The purpose of the frame procedures is to make the environment-manipulation procedures easy to read.

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

To look up a variable in an environment, we scan the list of variables in the first frame. If we find the desired variable, we return the corresponding element in the list of values. If we do not find the variable in the current frame, we search the enclosing environment, and so on. If we reach the empty environment, we signal an "unbound variable" error.

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

To set a variable to a new value in a specified environment, we scan for the variable, just as in `lookup-variable-value`, and change the corresponding value when we find it.

```
(define (set-variable-value! var val env)
  (define (env-loop env)
```

```
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable: SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

To define a variable, we search the first frame for a binding for the variable, and change the binding if it exists (just as in `set-variable-value!`). If no such binding exists, we adjoin one to the first frame.

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame) (frame-values frame))))
```

The method described here is only one of many plausible ways to represent environments. Since we used data abstraction to isolate the rest of the evaluator from the detailed choice of representation, we could change the environment representation if we wanted to. (See Exercise 4.11.) In a production-quality Lisp system, the speed of the evaluator's environment operations—especially that of variable lookup—has a major impact on the performance of the system. The representation described here, although conceptually simple, is not efficient and would

not ordinarily be used in a production system.[15]

Exercise 4.11: Instead of representing a frame as a pair of lists, we can represent a frame as a list of bindings, where each binding is a name-value pair. Rewrite the environment operations to use this alternative representation.

Exercise 4.12: The procedures `set-variable-value!`, `define-variable!` and `lookup-variable-value` can be expressed in terms of more abstract procedures for traversing the environment structure. Define abstractions that capture the common patterns and redefine the three procedures in terms of these abstractions.

Exercise 4.13: Scheme allows us to create new bindings for variables by means of `define`, but provides no way to get rid of bindings. Implement for the evaluator a special form `make-unbound!` that removes the binding of a given symbol from the environment in which the `make-unbound!` expression is evaluated. This problem is not completely specified. For example, should we remove only the binding in the first frame of the environment? Complete the specification and justify any choices you make.

[15] The drawback of this representation (as well as the variant in Exercise 4.11) is that the evaluator may have to search through many frames in order to find the binding for a given variable. (Such an approach is referred to as *deep binding*.) One way to avoid this inefficiency is to make use of a strategy called *lexical addressing*, which will be discussed in Section 5.5.6.

4.1.4 Running the Evaluator as a Program

Given the evaluator, we have in our hands a description (expressed in Lisp) of the process by which Lisp expressions are evaluated. One advantage of expressing the evaluator as a program is that we can run the program. This gives us, running within Lisp, a working model of how Lisp itself evaluates expressions. This can serve as a framework for experimenting with evaluation rules, as we shall do later in this chapter.

Our evaluator program reduces expressions ultimately to the application of primitive procedures. Therefore, all that we need to run the evaluator is to create a mechanism that calls on the underlying Lisp system to model the application of primitive procedures.

There must be a binding for each primitive procedure name, so that when `eval` evaluates the operator of an application of a primitive, it will find an object to pass to `apply`. We thus set up a global environment that associates unique objects with the names of the primitive procedures that can appear in the expressions we will be evaluating. The global environment also includes bindings for the symbols `true` and `false`, so that they can be used as variables in expressions to be evaluated.

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))
(define the-global-environment (setup-environment))
```

It does not matter how we represent the primitive procedure objects, so long as `apply` can identify and apply them by using the procedures

primitive-procedure? and apply-primitive-procedure. We have chosen to represent a primitive procedure as a list beginning with the symbol primitive and containing a procedure in the underlying Lisp that implements that primitive.

```
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))
(define (primitive-implementation proc) (cadr proc))
```

setup-environment will get the primitive names and implementation procedures from a list:[16]

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        ⟨more primitives⟩ ))
(define (primitive-procedure-names)
  (map car primitive-procedures))
(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))
```

To apply a primitive procedure, we simply apply the implementation procedure to the arguments, using the underlying Lisp system:[17]

[16] Any procedure defined in the underlying Lisp can be used as a primitive for the metacircular evaluator. The name of a primitive installed in the evaluator need not be the same as the name of its implementation in the underlying Lisp; the names are the same here because the metacircular evaluator implements Scheme itself. Thus, for example, we could put (list 'first car) or (list 'square (lambda (x) (* x x))) in the list of primitive-procedures.

[17] apply-in-underlying-scheme is the apply procedure we have used in earlier chapters. The metacircular evaluator's apply procedure (Section 4.1.1) models the

```
(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

For convenience in running the metacircular evaluator, we provide a *driver loop* that models the read-eval-print loop of the underlying Lisp system. It prints a *prompt*, reads an input expression, evaluates this expression in the global environment, and prints the result. We precede each printed result by an *output prompt* so as to distinguish the value of the expression from other output that may be printed.[18]

```
(define input-prompt  ";;; M-Eval input:")
(define output-prompt ";;; M-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

---

working of this primitive. Having two different things called `apply` leads to a technical problem in running the metacircular evaluator, because defining the metacircular evaluator's `apply` will mask the definition of the primitive. One way around this is to rename the metacircular `apply` to avoid conflict with the name of the primitive procedure. We have assumed instead that we have saved a reference to the underlying `apply` by doing

```
(define apply-in-underlying-scheme apply)
```

before defining the metacircular `apply`. This allows us to access the original version of `apply` under a different name.

[18] The primitive procedure `read` waits for input from the user, and returns the next complete expression that is typed. For example, if the user types `(+ 23 x)`, `read` returns a three-element list containing the symbol `+`, the number 23, and the symbol `x`. If the user types `'x`, `read` returns a two-element list containing the symbol `quote` and the symbol `x`.

```
(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))
(define (announce-output string)
  (newline) (display string) (newline))
```

We use a special printing procedure, `user-print`, to avoid printing the environment part of a compound procedure, which may be a very long list (or may even contain cycles).

```
(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))
```

Now all we need to do to run the evaluator is to initialize the global environment and start the driver loop. Here is a sample interaction:

```
(define the-global-environment (setup-environment))
(driver-loop)

;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
;;; M-Eval value:
ok
;;; M-Eval input:
(append '(a b c) '(d e f))
;;; M-Eval value:
(a b c d e f)
```

Exercise 4.14: Eva Lu Ator and Louis Reasoner are each experimenting with the metacircular evaluator. Eva types in the definition of `map`, and runs some test programs that use it. They work fine. Louis, in contrast, has installed the system version of `map` as a primitive for the metacircular evaluator. When he tries it, things go terribly wrong. Explain why Louis's `map` fails even though Eva's works.

4.1.5 Data as Programs

In thinking about a Lisp program that evaluates Lisp expressions, an analogy might be helpful. One operational view of the meaning of a program is that a program is a description of an abstract (perhaps infinitely large) machine. For example, consider the familiar program to compute factorials:

```
(define (factorial n)
  (if (= n 1) 1 (* (factorial (- n 1)) n)))
```

We may regard this program as the description of a machine containing parts that decrement, multiply, and test for equality, together with a two-position switch and another factorial machine. (The factorial machine is infinite because it contains another factorial machine within it.) Figure 4.2 is a flow diagram for the factorial machine, showing how the parts are wired together.

In a similar way, we can regard the evaluator as a very special machine that takes as input a description of a machine. Given this input, the evaluator configures itself to emulate the machine described. For example, if we feed our evaluator the definition of `factorial`, as shown in Figure 4.3, the evaluator will be able to compute factorials.

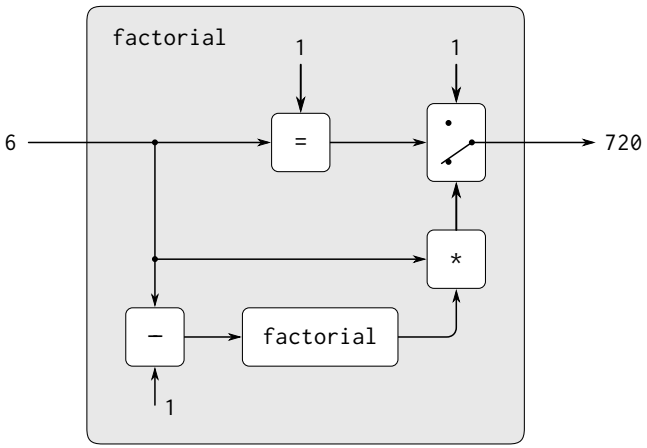

Figure 4.2: The factorial program, viewed as an abstract machine.

From this perspective, our evaluator is seen to be a *universal machine*. It mimics other machines when these are described as Lisp programs.[19] This is striking. Try to imagine an analogous evaluator for

[19]The fact that the machines are described in Lisp is inessential. If we give our evaluator a Lisp program that behaves as an evaluator for some other language, say C, the Lisp evaluator will emulate the C evaluator, which in turn can emulate any machine described as a C program. Similarly, writing a Lisp evaluator in C produces a C program that can execute any Lisp program. The deep idea here is that any evaluator can emulate any other. Thus, the notion of "what can in principle be computed" (ignoring practicalities of time and memory required) is independent of the language or the computer, and instead reflects an underlying notion of *computability*. This was first demonstrated in a clear way by Alan M. Turing (1912-1954), whose 1936 paper laid the foundations for theoretical computer science. In the paper, Turing presented a simple computational model—now known as a *Turing machine*—and argued that any "effective process" can be formulated as a program for such a machine. (This argument is known

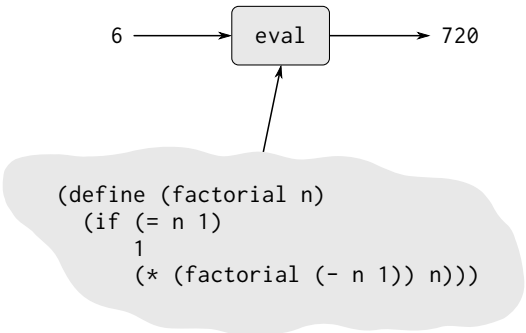

Figure 4.3: The evaluator emulating a factorial machine.

electrical circuits. This would be a circuit that takes as input a signal encoding the plans for some other circuit, such as a filter. Given this input, the circuit evaluator would then behave like a filter with the same description. Such a universal electrical circuit is almost unimaginably complex. It is remarkable that the program evaluator is a rather simple program.[20]

as the *Church-Turing thesis*.) Turing then implemented a universal machine, i.e., a Turing machine that behaves as an evaluator for Turing-machine programs. He used this framework to demonstrate that there are well-posed problems that cannot be computed by Turing machines (see Exercise 4.15), and so by implication cannot be formulated as "effective processes." Turing went on to make fundamental contributions to practical computer science as well. For example, he invented the idea of structuring programs using general-purpose subroutines. See Hodges 1983 for a biography of Turing.

[20] Some people find it counterintuitive that an evaluator, which is implemented by a relatively simple procedure, can emulate programs that are more complex than the evaluator itself. The existence of a universal evaluator machine is a deep and wonderful property of computation. *Recursion theory*, a branch of mathematical logic, is concerned with logical limits of computation. Douglas Hofstadter's beautiful book *Gödel, Escher, Bach* explores some of these ideas (Hofstadter 1979).

Another striking aspect of the evaluator is that it acts as a bridge between the data objects that are manipulated by our programming language and the programming language itself. Imagine that the evaluator program (implemented in Lisp) is running, and that a user is typing expressions to the evaluator and observing the results. From the perspective of the user, an input expression such as `(* x x)` is an expression in the programming language, which the evaluator should execute. From the perspective of the evaluator, however, the expression is simply a list (in this case, a list of three symbols: `*`, `x`, and `x`) that is to be manipulated according to a well-defined set of rules.

That the user's programs are the evaluator's data need not be a source of confusion. In fact, it is sometimes convenient to ignore this distinction, and to give the user the ability to explicitly evaluate a data object as a Lisp expression, by making `eval` available for use in programs. Many Lisp dialects provide a primitive `eval` procedure that takes as arguments an expression and an environment and evaluates the expression relative to the environment.[21] Thus,

```
(eval '(* 5 5) user-initial-environment)
```

and

```
(eval (cons '* (list 5 5)) user-initial-environment)
```

will both return 25.[22]

[21] Warning: This `eval` primitive is not identical to the `eval` procedure we implemented in Section 4.1.1, because it uses *actual* Scheme environments rather than the sample environment structures we built in Section 4.1.3. These actual environments cannot be manipulated by the user as ordinary lists; they must be accessed via `eval` or other special operations. Similarly, the `apply` primitive we saw earlier is not identical to the metacircular `apply`, because it uses actual Scheme procedures rather than the procedure objects we constructed in Section 4.1.3 and Section 4.1.4.

[22] The MIT implementation of Scheme includes `eval`, as well as a symbol `user-initial-environment` that is bound to the initial environment in which the user's in-

Exercise 4.15: Given a one-argument procedure `p` and an object `a`, `p` is said to "halt" on `a` if evaluating the expression `(p a)` returns a value (as opposed to terminating with an error message or running forever). Show that it is impossible to write a procedure `halts?` that correctly determines whether `p` halts on `a` for any procedure `p` and object `a`. Use the following reasoning: If you had such a procedure `halts?`, you could implement the following program:

```
(define (run-forever) (run-forever))
(define (try p)
  (if (halts? p p) (run-forever) 'halted))
```

Now consider evaluating the expression `(try try)` and show that any possible outcome (either halting or running forever) violates the intended behavior of `halts?`.[23]

4.1.6 Internal Definitions

Our environment model of evaluation and our metacircular evaluator execute definitions in sequence, extending the environment frame one definition at a time. This is particularly convenient for interactive program development, in which the programmer needs to freely mix the application of procedures with the definition of new procedures. However, if we think carefully about the internal definitions used to implement block structure (introduced in Section 1.1.8), we will find that

put expressions are evaluated.

[23] Although we stipulated that `halts?` is given a procedure object, notice that this reasoning still applies even if `halts?` can gain access to the procedure's text and its environment. This is Turing's celebrated *Halting Theorem*, which gave the first clear example of a *non-computable* problem, i.e., a well-posed task that cannot be carried out as a computational procedure.

name-by-name extension of the environment may not be the best way to define local variables.

Consider a procedure with internal definitions, such as

```
(define (f x)
  (define (even? n) (if (= n 0) true  (odd?  (- n 1))))
  (define (odd? n)  (if (= n 0) false (even? (- n 1))))
  ⟨rest of body of f⟩)
```

Our intention here is that the name `odd?` in the body of the procedure `even?` should refer to the procedure `odd?` that is defined after `even?`. The scope of the name `odd?` is the entire body of `f`, not just the portion of the body of `f` starting at the point where the `define` for `odd?` occurs. Indeed, when we consider that `odd?` is itself defined in terms of `even?`—so that `even?` and `odd?` are mutually recursive procedures—we see that the only satisfactory interpretation of the two `defines` is to regard them as if the names `even?` and `odd?` were being added to the environment simultaneously. More generally, in block structure, the scope of a local name is the entire procedure body in which the `define` is evaluated.

As it happens, our interpreter will evaluate calls to `f` correctly, but for an "accidental" reason: Since the definitions of the internal procedures come first, no calls to these procedures will be evaluated until all of them have been defined. Hence, `odd?` will have been defined by the time `even?` is executed. In fact, our sequential evaluation mechanism will give the same result as a mechanism that directly implements simultaneous definition for any procedure in which the internal definitions come first in a body and evaluation of the value expressions for the defined variables doesn't actually use any of the defined variables. (For an example of a procedure that doesn't obey these restrictions, so that sequential definition isn't equivalent to simultaneous definition,

see Exercise 4.19.)[24]

There is, however, a simple way to treat definitions so that internally defined names have truly simultaneous scope—just create all local variables that will be in the current environment before evaluating any of the value expressions. One way to do this is by a syntax transformation on `lambda` expressions. Before evaluating the body of a `lambda` expression, we "scan out" and eliminate all the internal definitions in the body. The internally defined variables will be created with a `let` and then set to their values by assignment. For example, the procedure

```
(lambda ⟨vars⟩
  (define u ⟨e1⟩)
  (define v ⟨e2⟩)
  ⟨e3⟩)
```

would be transformed into

```
(lambda ⟨vars⟩
  (let ((u '*unassigned*)
        (v '*unassigned*))
    (set! u ⟨e1⟩)
    (set! v ⟨e2⟩)
    ⟨e3⟩))
```

[24] Wanting programs to not depend on this evaluation mechanism is the reason for the "management is not responsible" remark in Footnote 28 of Chapter 1. By insisting that internal definitions come first and do not use each other while the definitions are being evaluated, the IEEE standard for Scheme leaves implementors some choice in the mechanism used to evaluate these definitions. The choice of one evaluation rule rather than another here may seem like a small issue, affecting only the interpretation of "badly formed" programs. However, we will see in Section 5.5.6 that moving to a model of simultaneous scoping for internal definitions avoids some nasty difficulties that would otherwise arise in implementing a compiler.

where `*unassigned*` is a special symbol that causes looking up a variable to signal an error if an attempt is made to use the value of the not-yet-assigned variable.

An alternative strategy for scanning out internal definitions is shown in Exercise 4.18. Unlike the transformation shown above, this enforces the restriction that the defined variables' values can be evaluated without using any of the variables' values.[25]

Exercise 4.16: In this exercise we implement the method just described for interpreting internal definitions. We assume that the evaluator supports `let` (see Exercise 4.6).

a. Change `lookup-variable-value` (Section 4.1.3) to signal an error if the value it finds is the symbol `*unassigned*`.

b. Write a procedure `scan-out-defines` that takes a procedure body and returns an equivalent one that has no internal definitions, by making the transformation described above.

c. Install `scan-out-defines` in the interpreter, either in `make-procedure` or in `procedure-body` (see Section 4.1.3). Which place is better? Why?

Exercise 4.17: Draw diagrams of the environment in effect when evaluating the expression ⟨*e3*⟩ in the procedure in the

[25] The IEEE standard for Scheme allows for different implementation strategies by specifying that it is up to the programmer to obey this restriction, not up to the implementation to enforce it. Some Scheme implementations, including MIT Scheme, use the transformation shown above. Thus, some programs that don't obey this restriction will in fact run in such implementations.

text, comparing how this will be structured when definitions are interpreted sequentially with how it will be structured if definitions are scanned out as described. Why is there an extra frame in the transformed program? Explain why this difference in environment structure can never make a difference in the behavior of a correct program. Design a way to make the interpreter implement the "simultaneous" scope rule for internal definitions without constructing the extra frame.

Exercise 4.18: Consider an alternative strategy for scanning out definitions that translates the example in the text to

```
(lambda ⟨vars⟩
  (let ((u '*unassigned*) (v '*unassigned*))
    (let ((a ⟨e1⟩) (b ⟨e2⟩))
      (set! u a)
      (set! v b))
    ⟨e3⟩))
```

Here `a` and `b` are meant to represent new variable names, created by the interpreter, that do not appear in the user's program. Consider the `solve` procedure from Section 3.5.4:

```
(define (solve f y0 dt)
  (define  y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

Will this procedure work if internal definitions are scanned out as shown in this exercise? What if they are scanned out as shown in the text? Explain.

Exercise 4.19: Ben Bitdiddle, Alyssa P. Hacker, and Eva Lu Ator are arguing about the desired result of evaluating the expression

```
(let ((a 1))
  (define (f x)
    (define b (+ a x))
    (define a 5)
    (+ a b))
  (f 10))
```

Ben asserts that the result should be obtained using the sequential rule for `define`: `b` is defined to be 11, then `a` is defined to be 5, so the result is 16. Alyssa objects that mutual recursion requires the simultaneous scope rule for internal procedure definitions, and that it is unreasonable to treat procedure names differently from other names. Thus, she argues for the mechanism implemented in Exercise 4.16. This would lead to `a` being unassigned at the time that the value for `b` is to be computed. Hence, in Alyssa's view the procedure should produce an error. Eva has a third opinion. She says that if the definitions of `a` and `b` are truly meant to be simultaneous, then the value 5 for `a` should be used in evaluating `b`. Hence, in Eva's view `a` should be 5, `b` should be 15, and the result should be 20. Which (if any) of these viewpoints do you support? Can you devise a way to implement internal definitions so that they behave as Eva prefers?[26]

[26] The MIT implementors of Scheme support Alyssa on the following grounds: Eva is in principle correct—the definitions should be regarded as simultaneous. But it seems difficult to implement a general, efficient mechanism that does what Eva requires. In the absence of such a mechanism, it is better to generate an error in the difficult cases of simultaneous definitions (Alyssa's notion) than to produce an incorrect answer (as Ben would have it).

Exercise 4.20: Because internal definitions look sequential but are actually simultaneous, some people prefer to avoid them entirely, and use the special form `letrec` instead. `letrec` looks like `let`, so it is not surprising that the variables it binds are bound simultaneously and have the same scope as each other. The sample procedure `f` above can be written without internal definitions, but with exactly the same meaning, as

```
(define (f x)
  (letrec
    ((even? (lambda (n)
              (if (= n 0) true  (odd?  (- n 1)))))
     (odd?  (lambda (n)
              (if (= n 0) false (even? (- n 1))))))
    ⟨rest of body of f⟩))
```

`letrec` expressions, which have the form

```
(letrec ((⟨var1⟩ ⟨exp1⟩) ... (⟨varn⟩ ⟨expn⟩))
  ⟨body⟩)
```

are a variation on `let` in which the expressions $\langle exp_k \rangle$ that provide the initial values for the variables $\langle var_k \rangle$ are evaluated in an environment that includes all the `letrec` bindings. This permits recursion in the bindings, such as the mutual recursion of `even?` and `odd?` in the example above, or the evaluation of 10 factorial with

```
(letrec
  ((fact (lambda (n)
           (if (= n 1) 1 (* n (fact (- n 1)))))))
  (fact 10))
```

a. Implement `letrec` as a derived expression, by transforming a `letrec` expression into a `let` expression as shown in the text above or in Exercise 4.18. That is, the `letrec` variables should be created with a `let` and then be assigned their values with `set!`.

b. Louis Reasoner is confused by all this fuss about internal definitions. The way he sees it, if you don't like to use `define` inside a procedure, you can just use `let`. Illustrate what is loose about his reasoning by drawing an environment diagram that shows the environment in which the ⟨*rest of body of f*⟩ is evaluated during evaluation of the expression `(f 5)`, with `f` defined as in this exercise. Draw an environment diagram for the same evaluation, but with `let` in place of `letrec` in the definition of `f`.

Exercise 4.21: Amazingly, Louis's intuition in Exercise 4.20 is correct. It is indeed possible to specify recursive procedures without using `letrec` (or even `define`), although the method for accomplishing this is much more subtle than Louis imagined. The following expression computes 10 factorial by applying a recursive factorial procedure:[27]

```
((lambda (n)
   ((lambda (fact) (fact fact n))
    (lambda (ft k) (if (= k 1) 1 (* k (ft ft (- k 1)))))))
 10)
```

[27] This example illustrates a programming trick for formulating recursive procedures without using `define`. The most general trick of this sort is the *Y operator*, which can be used to give a "pure λ-calculus" implementation of recursion. (See Stoy 1977 for details on the λ-calculus, and Gabriel 1988 for an exposition of the *Y* operator in Scheme.)

a. Check (by evaluating the expression) that this really does compute factorials. Devise an analogous expression for computing Fibonacci numbers.

b. Consider the following procedure, which includes mutually recursive internal definitions:

```
(define (f x)
  (define (even? n)
    (if (= n 0) true  (odd?  (- n 1))))
  (define (odd? n)
    (if (= n 0) false (even? (- n 1))))
  (even? x))
```

Fill in the missing expressions to complete an alternative definition of `f`, which uses neither internal definitions nor `letrec`:

```
(define (f x)
  ((lambda (even? odd?) (even? even? odd? x))
   (lambda (ev? od? n)
     (if (= n 0) true (od? ⟨??⟩ ⟨??⟩ ⟨??⟩)))
   (lambda (ev? od? n)
     (if (= n 0) false (ev? ⟨??⟩ ⟨??⟩ ⟨??⟩)))))
```

4.1.7 Separating Syntactic Analysis from Execution

The evaluator implemented above is simple, but it is very inefficient, because the syntactic analysis of expressions is interleaved with their execution. Thus if a program is executed many times, its syntax is analyzed many times. Consider, for example, evaluating `(factorial 4)` using the following definition of `factorial`:

```
(define (factorial n)
  (if (= n 1) 1 (* (factorial (- n 1)) n)))
```

Each time `factorial` is called, the evaluator must determine that the body is an `if` expression and extract the predicate. Only then can it evaluate the predicate and dispatch on its value. Each time it evaluates the expression `(* (factorial (- n 1)) n)`, or the subexpressions `(factorial (- n 1))` and `(- n 1)`, the evaluator must perform the case analysis in `eval` to determine that the expression is an application, and must extract its operator and operands. This analysis is expensive. Performing it repeatedly is wasteful.

We can transform the evaluator to be significantly more efficient by arranging things so that syntactic analysis is performed only once.[28] We split `eval`, which takes an expression and an environment, into two parts. The procedure `analyze` takes only the expression. It performs the syntactic analysis and returns a new procedure, the *execution procedure*, that encapsulates the work to be done in executing the analyzed expression. The execution procedure takes an environment as its argument and completes the evaluation. This saves work because `analyze` will be called only once on an expression, while the execution procedure may be called many times.

With the separation into analysis and execution, `eval` now becomes

```
(define (eval exp env) ((analyze exp) env))
```

The result of calling `analyze` is the execution procedure to be applied to the environment. The `analyze` procedure is the same case analysis as performed by the original `eval` of Section 4.1.1, except that the procedures to which we dispatch perform only analysis, not full evaluation:

```
(define (analyze exp)
```

[28] This technique is an integral part of the compilation process, which we shall discuss in Chapter 5. Jonathan Rees wrote a Scheme interpreter like this in about 1982 for the T project (Rees and Adams 1982). Marc Feeley (1986) (see also Feeley and Lapalme 1987) independently invented this technique in his master's thesis.

```
  (cond ((self-evaluating? exp) (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else (error "Unknown expression type: ANALYZE" exp))))
```

Here is the simplest syntactic analysis procedure, which handles self-evaluating expressions. It returns an execution procedure that ignores its environment argument and just returns the expression:

```
(define (analyze-self-evaluating exp)
  (lambda (env) exp))
```

For a quoted expression, we can gain a little efficiency by extracting the text of the quotation only once, in the analysis phase, rather than in the execution phase.

```
(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))
```

Looking up a variable value must still be done in the execution phase, since this depends upon knowing the environment.[29]

```
(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))
```

[29] There is, however, an important part of the variable search that *can* be done as part of the syntactic analysis. As we will show in Section 5.5.6, one can determine the position in the environment structure where the value of the variable will be found, thus obviating the need to scan the environment for the entry that matches the variable.

`analyze-assignment` also must defer actually setting the variable until the execution, when the environment has been supplied. However, the fact that the `assignment-value` expression can be analyzed (recursively) during analysis is a major gain in efficiency, because the `assignment-value` expression will now be analyzed only once. The same holds true for definitions.

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env)
      (set-variable-value! var (vproc env) env)
      'ok)))
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env)
      'ok)))
```

For `if` expressions, we extract and analyze the predicate, consequent, and alternative at analysis time.

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env) (if (true? (pproc env))
                      (cproc env)
                      (aproc env)))))
```

Analyzing a `lambda` expression also achieves a major gain in efficiency: We analyze the `lambda` body only once, even though procedures resulting from evaluation of the `lambda` may be applied many times.

```
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars bproc env))))
```

Analysis of a sequence of expressions (as in a `begin` or the body of a `lambda` expression) is more involved.[30] Each expression in the sequence is analyzed, yielding an execution procedure. These execution procedures are combined to produce an execution procedure that takes an environment as argument and sequentially calls each individual execution procedure with the environment as argument.

```
(define (analyze-sequence exps)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs) (error "Empty sequence: ANALYZE"))
    (loop (car procs) (cdr procs))))
```

To analyze an application, we analyze the operator and operands and construct an execution procedure that calls the operator execution procedure (to obtain the actual procedure to be applied) and the operand execution procedures (to obtain the actual arguments). We then pass these to `execute-application`, which is the analog of `apply` in Section 4.1.1. `execute-application` differs from `apply` in that the procedure body for a compound procedure has already been analyzed, so there is

[30] See Exercise 4.23 for some insight into the processing of sequences.

no need to do further analysis. Instead, we just call the execution procedure for the body on the extended environment.

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application
       (fproc env)
       (map (lambda (aproc) (aproc env))
            aprocs)))))
(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment
           (procedure-parameters proc)
           args
           (procedure-environment proc))))
        (else
         (error "Unknown procedure type: EXECUTE-APPLICATION"
                proc))))
```

Our new evaluator uses the same data structures, syntax procedures, and run-time support procedures as in sections Section 4.1.2, Section 4.1.3, and Section 4.1.4.

> **Exercise 4.22:** Extend the evaluator in this section to support the special form `let`. (See Exercise 4.6.)
>
> **Exercise 4.23:** Alyssa P. Hacker doesn't understand why `analyze-sequence` needs to be so complicated. All the other analysis procedures are straightforward transformations of

the corresponding evaluation procedures (or `eval` clauses) in Section 4.1.1. She expected `analyze-sequence` to look like this:

```
(define (analyze-sequence exps)
  (define (execute-sequence procs env)
    (cond ((null? (cdr procs))
           ((car procs) env))
          (else
           ((car procs) env)
           (execute-sequence (cdr procs) env))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence: ANALYZE"))
    (lambda (env)
      (execute-sequence procs env))))
```

Eva Lu Ator explains to Alyssa that the version in the text does more of the work of evaluating a sequence at analysis time. Alyssa's sequence-execution procedure, rather than having the calls to the individual execution procedures built in, loops through the procedures in order to call them: In effect, although the individual expressions in the sequence have been analyzed, the sequence itself has not been.

Compare the two versions of `analyze-sequence`. For example, consider the common case (typical of procedure bodies) where the sequence has just one expression. What work will the execution procedure produced by Alyssa's program do? What about the execution procedure produced by the program in the text above? How do the two versions compare for a sequence with two expressions?

Exercise 4.24: Design and carry out some experiments to compare the speed of the original metacircular evaluator with the version in this section. Use your results to estimate the fraction of time that is spent in analysis versus execution for various procedures.

4.2 Variations on a Scheme — Lazy Evaluation

Now that we have an evaluator expressed as a Lisp program, we can experiment with alternative choices in language design simply by modifying the evaluator. Indeed, new languages are often invented by first writing an evaluator that embeds the new language within an existing high-level language. For example, if we wish to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications. Not only does the high-level implementation base make it easier to test and debug the evaluator; in addition, the embedding enables the designer to snarf[31] features from the underlying language, just as our embedded Lisp evaluator uses primitives and control structure from the underlying Lisp. Only later (if ever) need the designer go to the trouble of building a complete implementation in a low-level language or in hardware. In this section and the next we explore some variations on Scheme that provide significant additional expressive power.

[31] Snarf: "To grab, especially a large document or file for the purpose of using it either with or without the owner's permission." Snarf down: "To snarf, sometimes with the connotation of absorbing, processing, or understanding." (These definitions were snarfed from Steele et al. 1983. See also Raymond 1993.)

4.2.1 Normal Order and Applicative Order

In Section 1.1, where we began our discussion of models of evaluation, we noted that Scheme is an *applicative-order* language, namely, that all the arguments to Scheme procedures are evaluated when the procedure is applied. In contrast, *normal-order* languages delay evaluation of procedure arguments until the actual argument values are needed. Delaying evaluation of procedure arguments until the last possible moment (e.g., until they are required by a primitive operation) is called *lazy evaluation*.[32] Consider the procedure

```
(define (try a b) (if (= a 0) 1 b))
```

Evaluating `(try 0 (/ 1 0))` generates an error in Scheme. With lazy evaluation, there would be no error. Evaluating the expression would return 1, because the argument `(/ 1 0)` would never be evaluated.

An example that exploits lazy evaluation is the definition of a procedure `unless`

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))
```

that can be used in expressions such as

```
(unless (= b 0)
        (/ a b)
        (begin (display "exception: returning 0") 0))
```

This won't work in an applicative-order language because both the usual value and the exceptional value will be evaluated before `unless` is called

[32] The difference between the "lazy" terminology and the "normal-order" terminology is somewhat fuzzy. Generally, "lazy" refers to the mechanisms of particular evaluators, while "normal-order" refers to the semantics of languages, independent of any particular evaluation strategy. But this is not a hard-and-fast distinction, and the two terminologies are often used interchangeably.

(compare Exercise 1.6). An advantage of lazy evaluation is that some procedures, such as `unless`, can do useful computation even if evaluation of some of their arguments would produce errors or would not terminate.

If the body of a procedure is entered before an argument has been evaluated we say that the procedure is *non-strict* in that argument. If the argument is evaluated before the body of the procedure is entered we say that the procedure is *strict* in that argument.[33] In a purely applicative-order language, all procedures are strict in each argument. In a purely normal-order language, all compound procedures are non-strict in each argument, and primitive procedures may be either strict or non-strict. There are also languages (see Exercise 4.31) that give programmers detailed control over the strictness of the procedures they define.

A striking example of a procedure that can usefully be made non-strict is `cons` (or, in general, almost any constructor for data structures). One can do useful computation, combining elements to form data structures and operating on the resulting data structures, even if the values of the elements are not known. It makes perfect sense, for instance, to compute the length of a list without knowing the values of the individual elements in the list. We will exploit this idea in Section 4.2.3 to implement the streams of Chapter 3 as lists formed of non-strict `cons` pairs.

> **Exercise 4.25:** Suppose that (in ordinary applicative-order Scheme) we define `unless` as shown above and then define

[33] The "strict" versus "non-strict" terminology means essentially the same thing as "applicative-order" versus "normal-order," except that it refers to individual procedures and arguments rather than to the language as a whole. At a conference on programming languages you might hear someone say, "The normal-order language Hassle has certain strict primitives. Other procedures take their arguments by lazy evaluation."

`factorial` in terms of `unless` as

```
(define (factorial n)
  (unless (= n 1)
          (* n (factorial (- n 1)))
          1))
```

What happens if we attempt to evaluate `(factorial 5)`? Will our definitions work in a normal-order language?

Exercise 4.26: Ben Bitdiddle and Alyssa P. Hacker disagree over the importance of lazy evaluation for implementing things such as `unless`. Ben points out that it's possible to implement `unless` in applicative order as a special form. Alyssa counters that, if one did that, `unless` would be merely syntax, not a procedure that could be used in conjunction with higher-order procedures. Fill in the details on both sides of the argument. Show how to implement `unless` as a derived expression (like `cond` or `let`), and give an example of a situation where it might be useful to have `unless` available as a procedure, rather than as a special form.

4.2.2 An Interpreter with Lazy Evaluation

In this section we will implement a normal-order language that is the same as Scheme except that compound procedures are non-strict in each argument. Primitive procedures will still be strict. It is not difficult to modify the evaluator of Section 4.1.1 so that the language it interprets behaves this way. Almost all the required changes center around procedure application.

The basic idea is that, when applying a procedure, the interpreter must determine which arguments are to be evaluated and which are to

be delayed. The delayed arguments are not evaluated; instead, they are transformed into objects called *thunks*.[34] The thunk must contain the information required to produce the value of the argument when it is needed, as if it had been evaluated at the time of the application. Thus, the thunk must contain the argument expression and the environment in which the procedure application is being evaluated.

The process of evaluating the expression in a thunk is called *forcing*.[35] In general, a thunk will be forced only when its value is needed: when it is passed to a primitive procedure that will use the value of the thunk; when it is the value of a predicate of a conditional; and when it is the value of an operator that is about to be applied as a procedure. One design choice we have available is whether or not to *memoize* thunks, as we did with delayed objects in Section 3.5.1. With memoization, the first time a thunk is forced, it stores the value that is computed. Subsequent forcings simply return the stored value without repeating the computation. We'll make our interpreter memoize, because this is more efficient for many applications. There are tricky considerations here, however.[36]

[34]The word *thunk* was invented by an informal working group that was discussing the implementation of call-by-name in Algol 60. They observed that most of the analysis of ("thinking about") the expression could be done at compile time; thus, at run time, the expression would already have been "thunk" about (Ingerman et al. 1960).

[35]This is analogous to the use of `force` on the delayed objects that were introduced in Chapter 3 to represent streams. The critical difference between what we are doing here and what we did in Chapter 3 is that we are building delaying and forcing into the evaluator, and thus making this uniform and automatic throughout the language.

[36]Lazy evaluation combined with memoization is sometimes referred to as *call-by-need* argument passing, in contrast to *call-by-name* argument passing. (Call-by-name, introduced in Algol 60, is similar to non-memoized lazy evaluation.) As language designers, we can build our evaluator to memoize, not to memoize, or leave this an option for programmers (Exercise 4.31). As you might expect from Chapter 3, these choices raise issues that become both subtle and confusing in the presence of assignments. (See Exercise 4.27 and Exercise 4.29.) An excellent article by Clinger (1982) attempts to clar-

Modifying the evaluator

The main difference between the lazy evaluator and the one in Section 4.1 is in the handling of procedure applications in `eval` and `apply`.

The `application?` clause of `eval` becomes

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

This is almost the same as the `application?` clause of `eval` in Section 4.1.1. For lazy evaluation, however, we call `apply` with the operand expressions, rather than the arguments produced by evaluating them. Since we will need the environment to construct thunks if the arguments are to be delayed, we must pass this as well. We still evaluate the operator, because `apply` needs the actual procedure to be applied in order to dispatch on its type (primitive versus compound) and apply it.

Whenever we need the actual value of an expression, we use

```
(define (actual-value exp env)
  (force-it (eval exp env)))
```

instead of just `eval`, so that if the expression's value is a thunk, it will be forced.

Our new version of `apply` is also almost the same as the version in Section 4.1.1. The difference is that `eval` has passed in unevaluated operand expressions: For primitive procedures (which are strict), we evaluate all the arguments before applying the primitive; for compound procedures (which are non-strict) we delay all the arguments before applying the procedure.

---

ify the multiple dimensions of confusion that arise here.

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))    ; changed
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)  ; changed
           (procedure-environment procedure))))
        (else (error "Unknown procedure type: APPLY"
                     procedure))))
```

The procedures that process the arguments are just like `list-of-values` from Section 4.1.1, except that `list-of-delayed-args` delays the arguments instead of evaluating them, and `list-of-arg-values` uses `actual-value` instead of `eval`:

```
(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps)
                          env)
            (list-of-arg-values (rest-operands exps)
                                env))))
(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps)
                      env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))
```

The other place we must change the evaluator is in the handling of `if`, where we must use `actual-value` instead of `eval` to get the value of the predicate expression before testing whether it is true or false:

```
(define (eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Finally, we must change the `driver-loop` procedure (Section 4.1.4) to use `actual-value` instead of `eval`, so that if a delayed value is propagated back to the read-eval-print loop, it will be forced before being printed. We also change the prompts to indicate that this is the lazy evaluator:

```
(define input-prompt  ";;; L-Eval input:")
(define output-prompt ";;; L-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
           (actual-value
            input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

With these changes made, we can start the evaluator and test it. The successful evaluation of the `try` expression discussed in Section 4.2.1 indicates that the interpreter is performing lazy evaluation:

```
(define the-global-environment (setup-environment))
(driver-loop)
;;; L-Eval input:
(define (try a b) (if (= a 0) 1 b))
```

```
;;; L-Eval value:
ok
;;; L-Eval input:
(try 0 (/ 1 0))
;;; L-Eval value:
1
```

Representing thunks

Our evaluator must arrange to create thunks when procedures are applied to arguments and to force these thunks later. A thunk must package an expression together with the environment, so that the argument can be produced later. To force the thunk, we simply extract the expression and environment from the thunk and evaluate the expression in the environment. We use `actual-value` rather than `eval` so that in case the value of the expression is itself a thunk, we will force that, and so on, until we reach something that is not a thunk:

```
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

One easy way to package an expression with an environment is to make a list containing the expression and the environment. Thus, we create a thunk as follows:

```
(define (delay-it exp env)
  (list 'thunk exp env))
(define (thunk? obj)
  (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr  thunk))
(define (thunk-env thunk) (caddr thunk))
```

Actually, what we want for our interpreter is not quite this, but rather thunks that have been memoized. When a thunk is forced, we will turn it into an evaluated thunk by replacing the stored expression with its value and changing the `thunk` tag so that it can be recognized as already evaluated.[37]

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj)
                                     (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj)
                     result)      ; replace exp with its value
           (set-cdr! (cdr obj)
                     '())         ; forget unneeded env
           result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))
```

Notice that the same `delay-it` procedure works both with and without memoization.

[37]Notice that we also erase the `env` from the thunk once the expression's value has been computed. This makes no difference in the values returned by the interpreter. It does help save space, however, because removing the reference from the thunk to the `env` once it is no longer needed allows this structure to be *garbage-collected* and its space recycled, as we will discuss in Section 5.3.

Similarly, we could have allowed unneeded environments in the memoized delayed objects of Section 3.5.1 to be garbage-collected, by having `memo-proc` do something like `(set! proc '())` to discard the procedure `proc` (which includes the environment in which the `delay` was evaluated) after storing its value.

Exercise 4.27: Suppose we type in the following definitions to the lazy evaluator:

```
(define count 0)
(define (id x) (set! count (+ count 1)) x)
```

Give the missing values in the following sequence of interactions, and explain your answers.[38]

```
(define w (id (id 10)))
;;; L-Eval input:
count
;;; L-Eval value:
⟨response⟩
;;; L-Eval input:
w
;;; L-Eval value:
⟨response⟩
;;; L-Eval input:
count
;;; L-Eval value:
⟨response⟩
```

Exercise 4.28: `eval` uses `actual-value` rather than `eval` to evaluate the operator before passing it to `apply`, in order to force the value of the operator. Give an example that demonstrates the need for this forcing.

Exercise 4.29: Exhibit a program that you would expect to run much more slowly without memoization than with

[38] This exercise demonstrates that the interaction between lazy evaluation and side effects can be very confusing. This is just what you might expect from the discussion in Chapter 3.

memoization. Also, consider the following interaction, where the `id` procedure is defined as in Exercise 4.27 and `count` starts at 0:

```
(define (square x) (* x x))
;;; L-Eval input:
(square (id 10))
;;; L-Eval value:
⟨response⟩
;;; L-Eval input:
count
;;; L-Eval value:
⟨response⟩
```

Give the responses both when the evaluator memoizes and when it does not.

Exercise 4.30: Cy D. Fect, a reformed C programmer, is worried that some side effects may never take place, because the lazy evaluator doesn't force the expressions in a sequence. Since the value of an expression in a sequence other than the last one is not used (the expression is there only for its effect, such as assigning to a variable or printing), there can be no subsequent use of this value (e.g., as an argument to a primitive procedure) that will cause it to be forced. Cy thus thinks that when evaluating sequences, we must force all expressions in the sequence except the final one. He proposes to modify `eval-sequence` from Section 4.1.1 to use `actual-value` rather than `eval`:

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (actual-value (first-exp exps) env)
```

```
          (eval-sequence (rest-exps exps) env))))
```

a. Ben Bitdiddle thinks Cy is wrong. He shows Cy the `for-each` procedure described in Exercise 2.23, which gives an important example of a sequence with side effects:

```
(define (for-each proc items)
  (if (null? items)
      'done
      (begin (proc (car items))
             (for-each proc (cdr items)))))
```

He claims that the evaluator in the text (with the original `eval-sequence`) handles this correctly:

```
;;; L-Eval input:
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
57
321
88
;;; L-Eval value:
done
```

Explain why Ben is right about the behavior of `for-each`.

b. Cy agrees that Ben is right about the `for-each` example, but says that that's not the kind of program he was thinking about when he proposed his change to `eval-sequence`. He defines the following two procedures in the lazy evaluator:

```
(define (p1 x)
  (set! x (cons x '(2)))
  x)
(define (p2 x)
  (define (p e)
    e
    x)
  (p (set! x (cons x '(2)))))
```

What are the values of `(p1 1)` and `(p2 1)` with the original `eval-sequence`? What would the values be with Cy's proposed change to `eval-sequence`?

c. Cy also points out that changing `eval-sequence` as he proposes does not affect the behavior of the example in part a. Explain why this is true.

d. How do you think sequences ought to be treated in the lazy evaluator? Do you like Cy's approach, the approach in the text, or some other approach?

Exercise 4.31: The approach taken in this section is somewhat unpleasant, because it makes an incompatible change to Scheme. It might be nicer to implement lazy evaluation as an *upward-compatible extension*, that is, so that ordinary Scheme programs will work as before. We can do this by extending the syntax of procedure declarations to let the user control whether or not arguments are to be delayed. While we're at it, we may as well also give the user the choice between delaying with and without memoization. For example, the definition

```
(define (f a (b lazy) c (d lazy-memo))
  ...)
```

would define `f` to be a procedure of four arguments, where the first and third arguments are evaluated when the procedure is called, the second argument is delayed, and the fourth argument is both delayed and memoized. Thus, ordinary procedure definitions will produce the same behavior as ordinary Scheme, while adding the `lazy-memo` declaration to each parameter of every compound procedure will produce the behavior of the lazy evaluator defined in this section. Design and implement the changes required to produce such an extension to Scheme. You will have to implement new syntax procedures to handle the new syntax for `define`. You must also arrange for `eval` or `apply` to determine when arguments are to be delayed, and to force or delay arguments accordingly, and you must arrange for forcing to memoize or not, as appropriate.

4.2.3 Streams as Lazy Lists

In Section 3.5.1, we showed how to implement streams as delayed lists. We introduced special forms `delay` and `cons-stream`, which allowed us to construct a "promise" to compute the `cdr` of a stream, without actually fulfilling that promise until later. We could use this general technique of introducing special forms whenever we need more control over the evaluation process, but this is awkward. For one thing, a special form is not a first-class object like a procedure, so we cannot use it together with higher-order procedures.[39] Additionally, we were forced to create streams as a new kind of data object similar but not identical to lists, and this required us to reimplement many ordinary list operations

[39] This is precisely the issue with the `unless` procedure, as in Exercise 4.26.

(`map`, `append`, and so on) for use with streams.

With lazy evaluation, streams and lists can be identical, so there is no need for special forms or for separate list and stream operations. All we need to do is to arrange matters so that `cons` is non-strict. One way to accomplish this is to extend the lazy evaluator to allow for non-strict primitives, and to implement `cons` as one of these. An easier way is to recall (Section 2.1.3) that there is no fundamental need to implement `cons` as a primitive at all. Instead, we can represent pairs as procedures:[40]

```
(define (cons x y) (lambda (m) (m x y)))
(define (car z) (z (lambda (p q) p)))
(define (cdr z) (z (lambda (p q) q)))
```

In terms of these basic operations, the standard definitions of the list operations will work with infinite lists (streams) as well as finite ones, and the stream operations can be implemented as list operations. Here are some examples:

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items)) (map proc (cdr items)))))
(define (scale-list items factor)
  (map (lambda (x) (* x factor)) items))
```

[40] This is the procedural representation described in Exercise 2.4. Essentially any procedural representation (e.g., a message-passing implementation) would do as well. Notice that we can install these definitions in the lazy evaluator simply by typing them at the driver loop. If we had originally included `cons`, `car`, and `cdr` as primitives in the global environment, they will be redefined. (Also see Exercise 4.33 and Exercise 4.34.)

```
(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))
(define ones (cons 1 ones))
(define integers (cons 1 (add-lists ones integers)))
;;; L-Eval input:
(list-ref integers 17)
;;; L-Eval value:
18
```

Note that these lazy lists are even lazier than the streams of Chapter 3: The `car` of the list, as well as the `cdr`, is delayed.[41] In fact, even accessing the `car` or `cdr` of a lazy pair need not force the value of a list element. The value will be forced only when it is really needed—e.g., for use as the argument of a primitive, or to be printed as an answer.

Lazy pairs also help with the problem that arose with streams in Section 3.5.4, where we found that formulating stream models of systems with loops may require us to sprinkle our programs with explicit `delay` operations, beyond the ones supplied by `cons-stream`. With lazy evaluation, all arguments to procedures are delayed uniformly. For instance, we can implement procedures to integrate lists and solve differential equations as we originally intended in Section 3.5.4:

```
(define (integral integrand initial-value dt)
  (define int
    (cons initial-value
          (add-lists (scale-list integrand dt) int)))
  int)
```

[41] This permits us to create delayed versions of more general kinds of list structures, not just sequences. Hughes 1990 discusses some applications of "lazy trees."

```
(define (solve f y0 dt)
  (define  y (integral dy y0 dt))
  (define dy (map f y))
  y)
;;; L-Eval input:
(list-ref (solve (lambda (x) x) 1 0.001) 1000)
;;; L-Eval value:
2.716924
```

Exercise 4.32: Give some examples that illustrate the difference between the streams of Chapter 3 and the "lazier" lazy lists described in this section. How can you take advantage of this extra laziness?

Exercise 4.33: Ben Bitdiddle tests the lazy list implementation given above by evaluating the expression:

```
(car '(a b c))
```

To his surprise, this produces an error. After some thought, he realizes that the "lists" obtained by reading in quoted expressions are different from the lists manipulated by the new definitions of `cons`, `car`, and `cdr`. Modify the evaluator's treatment of quoted expressions so that quoted lists typed at the driver loop will produce true lazy lists.

Exercise 4.34: Modify the driver loop for the evaluator so that lazy pairs and lists will print in some reasonable way. (What are you going to do about infinite lists?) You may also need to modify the representation of lazy pairs so that the evaluator can identify them in order to print them.

4.3 Variations on a Scheme — Nondeterministic Computing

In this section, we extend the Scheme evaluator to support a programming paradigm called *nondeterministic computing* by building into the evaluator a facility to support automatic search. This is a much more profound change to the language than the introduction of lazy evaluation in Section 4.2.

Nondeterministic computing, like stream processing, is useful for "generate and test" applications. Consider the task of starting with two lists of positive integers and finding a pair of integers—one from the first list and one from the second list—whose sum is prime. We saw how to handle this with finite sequence operations in Section 2.2.3 and with infinite streams in Section 3.5.3. Our approach was to generate the sequence of all possible pairs and filter these to select the pairs whose sum is prime. Whether we actually generate the entire sequence of pairs first as in Chapter 2, or interleave the generating and filtering as in Chapter 3, is immaterial to the essential image of how the computation is organized.

The nondeterministic approach evokes a different image. Imagine simply that we choose (in some way) a number from the first list and a number from the second list and require (using some mechanism) that their sum be prime. This is expressed by following procedure:

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

It might seem as if this procedure merely restates the problem, rather than specifying a way to solve it. Nevertheless, this is a legitimate non-

deterministic program.[42]

The key idea here is that expressions in a nondeterministic language can have more than one possible value. For instance, `an-element-of` might return any element of the given list. Our nondeterministic program evaluator will work by automatically choosing a possible value and keeping track of the choice. If a subsequent requirement is not met, the evaluator will try a different choice, and it will keep trying new choices until the evaluation succeeds, or until we run out of choices. Just as the lazy evaluator freed the programmer from the details of how values are delayed and forced, the nondeterministic program evaluator will free the programmer from the details of how choices are made.

It is instructive to contrast the different images of time evoked by nondeterministic evaluation and stream processing. Stream processing uses lazy evaluation to decouple the time when the stream of possible answers is assembled from the time when the actual stream elements are produced. The evaluator supports the illusion that all the possible answers are laid out before us in a timeless sequence. With nondeterministic evaluation, an expression represents the exploration of a set of possible worlds, each determined by a set of choices. Some of the possible worlds lead to dead ends, while others have useful values. The nondeterministic program evaluator supports the illusion that time branches, and that our programs have different possible execution histories. When

[42] We assume that we have previously defined a procedure `prime?` that tests whether numbers are prime. Even with `prime?` defined, the `prime-sum-pair` procedure may look suspiciously like the unhelpful "pseudo-Lisp" attempt to define the square-root function, which we described at the beginning of Section 1.1.7. In fact, a square-root procedure along those lines can actually be formulated as a nondeterministic program. By incorporating a search mechanism into the evaluator, we are eroding the distinction between purely declarative descriptions and imperative specifications of how to compute answers. We'll go even farther in this direction in Section 4.4.

we reach a dead end, we can revisit a previous choice point and proceed along a different branch.

The nondeterministic program evaluator implemented below is called the `amb` evaluator because it is based on a new special form called `amb`. We can type the above definition of `prime-sum-pair` at the `amb` evaluator driver loop (along with definitions of `prime?`, `an-element-of`, and `require`) and run the procedure as follows:

```
;;; Amb-Eval input:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Starting a new problem
;;; Amb-Eval value:
(3 20)
```

The value returned was obtained after the evaluator repeatedly chose elements from each of the lists, until a successful choice was made.

Section 4.3.1 introduces `amb` and explains how it supports nondeterminism through the evaluator's automatic search mechanism. Section 4.3.2 presents examples of nondeterministic programs, and Section 4.3.3 gives the details of how to implement the `amb` evaluator by modifying the ordinary Scheme evaluator.

4.3.1 Amb and Search

To extend Scheme to support nondeterminism, we introduce a new special form called `amb`.[43] The expression

`(amb` $\langle e_1 \rangle$ $\langle e_2 \rangle$ ... $\langle e_n \rangle$`)`

returns the value of one of the n expressions $\langle e_i \rangle$ "ambiguously." For example, the expression

[43] The idea of `amb` for nondeterministic programming was first described in 1961 by John McCarthy (see McCarthy 1963).

```
(list (amb 1 2 3) (amb 'a 'b))
```

can have six possible values:

```
(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)
```

`amb` with a single choice produces an ordinary (single) value.

`amb` with no choices—the expression `(amb)`—is an expression with no acceptable values. Operationally, we can think of `(amb)` as an expression that when evaluated causes the computation to "fail": The computation aborts and no value is produced. Using this idea, we can express the requirement that a particular predicate expression `p` must be true as follows:

```
(define (require p) (if (not p) (amb)))
```

With `amb` and `require`, we can implement the `an-element-of` procedure used above:

```
(define (an-element-of items)
  (require (not (null? items)))
  (amb (car items) (an-element-of (cdr items))))
```

`an-element-of` fails if the list is empty. Otherwise it ambiguously returns either the first element of the list or an element chosen from the rest of the list.

We can also express infinite ranges of choices. The following procedure potentially returns any integer greater than or equal to some given *n*:

```
(define (an-integer-starting-from n)
  (amb n (an-integer-starting-from (+ n 1))))
```

This is like the stream procedure `integers-starting-from` described in Section 3.5.2, but with an important difference: The stream procedure

returns an object that represents the sequence of all integers beginning with n, whereas the `amb` procedure returns a single integer.[44]

Abstractly, we can imagine that evaluating an `amb` expression causes time to split into branches, where the computation continues on each branch with one of the possible values of the expression. We say that `amb` represents a *nondeterministic choice point*. If we had a machine with a sufficient number of processors that could be dynamically allocated, we could implement the search in a straightforward way. Execution would proceed as in a sequential machine, until an `amb` expression is encountered. At this point, more processors would be allocated and initialized to continue all of the parallel executions implied by the choice. Each processor would proceed sequentially as if it were the only choice, until it either terminates by encountering a failure, or it further subdivides, or it finishes.[45]

On the other hand, if we have a machine that can execute only one process (or a few concurrent processes), we must consider the alternatives sequentially. One could imagine modifying an evaluator to pick at random a branch to follow whenever it encounters a choice point.

[44] In actuality, the distinction between nondeterministically returning a single choice and returning all choices depends somewhat on our point of view. From the perspective of the code that uses the value, the nondeterministic choice returns a single value. From the perspective of the programmer designing the code, the nondeterministic choice potentially returns all possible values, and the computation branches so that each value is investigated separately.

[45] One might object that this is a hopelessly inefficient mechanism. It might require millions of processors to solve some easily stated problem this way, and most of the time most of those processors would be idle. This objection should be taken in the context of history. Memory used to be considered just such an expensive commodity. In 1964 a megabyte of RAM cost about $400,000. Now every personal computer has many megabytes of RAM, and most of the time most of that RAM is unused. It is hard to underestimate the cost of mass-produced electronics.

Random choice, however, can easily lead to failing values. We might try running the evaluator over and over, making random choices and hoping to find a non-failing value, but it is better to *systematically search* all possible execution paths. The `amb` evaluator that we will develop and work with in this section implements a systematic search as follows: When the evaluator encounters an application of `amb`, it initially selects the first alternative. This selection may itself lead to a further choice. The evaluator will always initially choose the first alternative at each choice point. If a choice results in a failure, then the evaluator automagically[46] *backtracks* to the most recent choice point and tries the next alternative. If it runs out of alternatives at any choice point, the evaluator will back up to the previous choice point and resume from there. This process leads to a search strategy known as *depth-first search* or *chronological backtracking*.[47]

[46] Automagically: "Automatically, but in a way which, for some reason (typically because it is too complicated, or too ugly, or perhaps even too trivial), the speaker doesn't feel like explaining." (Steele et al. 1983, Raymond 1993)

[47] The integration of automatic search strategies into programming languages has had a long and checkered history. The first suggestions that nondeterministic algorithms might be elegantly encoded in a programming language with search and automatic backtracking came from Robert Floyd (1967). Carl Hewitt (1969) invented a programming language called Planner that explicitly supported automatic chronological backtracking, providing for a built-in depth-first search strategy. Sussman et al. (1971) implemented a subset of this language, called MicroPlanner, which was used to support work in problem solving and robot planning. Similar ideas, arising from logic and theorem proving, led to the genesis in Edinburgh and Marseille of the elegant language Prolog (which we will discuss in Section 4.4). After sufficient frustration with automatic search, McDermott and Sussman (1972) developed a language called Conniver, which included mechanisms for placing the search strategy under programmer control. This proved unwieldy, however, and Sussman and Stallman 1975 found a more tractable approach while investigating methods of symbolic analysis for electrical circuits. They developed a non-chronological backtracking scheme that was based on

Driver loop

The driver loop for the `amb` evaluator has some unusual properties. It reads an expression and prints the value of the first non-failing execution, as in the `prime-sum-pair` example shown above. If we want to see the value of the next successful execution, we can ask the interpreter to backtrack and attempt to generate a second non-failing execution. This is signaled by typing the symbol `try-again`. If any expression except `try-again` is given, the interpreter will start a new problem, discarding the unexplored alternatives in the previous problem. Here is a sample interaction:

```
;;; Amb-Eval input:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Starting a new problem
;;; Amb-Eval value:
(3 20)

;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
```

tracing out the logical dependencies connecting facts, a technique that has come to be known as *dependency-directed backtracking*. Although their method was complex, it produced reasonably efficient programs because it did little redundant search. Doyle (1979) and McAllester (1978; 1980) generalized and clarified the methods of Stallman and Sussman, developing a new paradigm for formulating search that is now called *truth maintenance*. Modern problem-solving systems all use some form of truth-maintenance system as a substrate. See Forbus and deKleer 1993 for a discussion of elegant ways to build truth-maintenance systems and applications using truth maintenance. Zabih et al. 1987 describes a nondeterministic extension to Scheme that is based on `amb`; it is similar to the interpreter described in this section, but more sophisticated, because it uses dependency-directed backtracking rather than chronological backtracking. Winston 1992 gives an introduction to both kinds of backtracking.

```
(3 110)

;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
(8 35)

;;; Amb-Eval input:
try-again
;;; There are no more values of
(prime-sum-pair (quote (1 3 5 8)) (quote (20 35 110)))

;;; Amb-Eval input:
(prime-sum-pair '(19 27 30) '(11 36 58))
;;; Starting a new problem
;;; Amb-Eval value:
(30 11)
```

Exercise 4.35: Write a procedure `an-integer-between` that returns an integer between two given bounds. This can be used to implement a procedure that finds Pythagorean triples, i.e., triples of integers (i, j, k) between the given bounds such that $i \leq j$ and $i^2 + j^2 = k^2$, as follows:

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

Exercise 4.36: Exercise 3.69 discussed how to generate the stream of *all* Pythagorean triples, with no upper bound on

the size of the integers to be searched. Explain why simply replacing `an-integer-between` by `an-integer-starting-from` in the procedure in Exercise 4.35 is not an adequate way to generate arbitrary Pythagorean triples. Write a procedure that actually will accomplish this. (That is, write a procedure for which repeatedly typing `try-again` would in principle eventually generate all Pythagorean triples.)

Exercise 4.37: Ben Bitdiddle claims that the following method for generating Pythagorean triples is more efficient than the one in Exercise 4.35. Is he correct? (Hint: Consider the number of possibilities that must be explored.)

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high))
        (hsq (* high high)))
    (let ((j (an-integer-between i high)))
      (let ((ksq (+ (* i i) (* j j))))
        (require (>= hsq ksq))
        (let ((k (sqrt ksq)))
          (require (integer? k))
          (list i j k))))))
```

4.3.2 Examples of Nondeterministic Programs

Section 4.3.3 describes the implementation of the `amb` evaluator. First, however, we give some examples of how it can be used. The advantage of nondeterministic programming is that we can suppress the details of how search is carried out, thereby expressing our programs at a higher level of abstraction.

Logic Puzzles

The following puzzle (taken from Dinesman 1968) is typical of a large class of simple logic puzzles:

> Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?

We can determine who lives on each floor in a straightforward way by enumerating all the possibilities and imposing the given restrictions:[48]

```
(define (multiple-dwelling)
  (let ((baker    (amb 1 2 3 4 5)) (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5)) (miller (amb 1 2 3 4 5))
        (smith    (amb 1 2 3 4 5)))
    (require
     (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
```

[48] Our program uses the following procedure to determine if the elements of a list are distinct:

```
(define (distinct? items)
  (cond ((null? items) true)
        ((null? (cdr items)) true)
        ((member (car items) (cdr items)) false)
        (else (distinct? (cdr items)))))
```

`member` is like `memq` except that it uses `equal?` instead of `eq?` to test for equality.

```
(require (not (= cooper 1)))
(require (not (= fletcher 5)))
(require (not (= fletcher 1)))
(require (> miller cooper))
(require (not (= (abs (- smith fletcher)) 1)))
(require (not (= (abs (- fletcher cooper)) 1)))
(list (list 'baker baker)       (list 'cooper cooper)
      (list 'fletcher fletcher) (list 'miller miller)
      (list 'smith smith))))
```

Evaluating the expression `(multiple-dwelling)` produces the result

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

Although this simple procedure works, it is very slow. Exercise 4.39 and Exercise 4.40 discuss some possible improvements.

> **Exercise 4.38:** Modify the multiple-dwelling procedure to omit the requirement that Smith and Fletcher do not live on adjacent floors. How many solutions are there to this modified puzzle?
>
> **Exercise 4.39:** Does the order of the restrictions in the multiple-dwelling procedure affect the answer? Does it affect the time to find an answer? If you think it matters, demonstrate a faster program obtained from the given one by reordering the restrictions. If you think it does not matter, argue your case.
>
> **Exercise 4.40:** In the multiple dwelling problem, how many sets of assignments are there of people to floors, both before and after the requirement that floor assignments be distinct? It is very inefficient to generate all possible assignments of people to floors and then leave it to backtracking

to eliminate them. For example, most of the restrictions depend on only one or two of the person-floor variables, and can thus be imposed before floors have been selected for all the people. Write and demonstrate a much more efficient nondeterministic procedure that solves this problem based upon generating only those possibilities that are not already ruled out by previous restrictions. (Hint: This will require a nest of `let` expressions.)

Exercise 4.41: Write an ordinary Scheme program to solve the multiple dwelling puzzle.

Exercise 4.42: Solve the following "Liars" puzzle (from Phillips 1934):

Five schoolgirls sat for an examination. Their parents—so they thought—showed an undue degree of interest in the result. They therefore agreed that, in writing home about the examination, each girl should make one true statement and one untrue one. The following are the relevant passages from their letters:

- Betty: "Kitty was second in the examination. I was only third."
- Ethel: "You'll be glad to hear that I was on top. Joan was 2nd."
- Joan: "I was third, and poor old Ethel was bottom."
- Kitty: "I came out second. Mary was only fourth."
- Mary: "I was fourth. Top place was taken by Betty."

What in fact was the order in which the five girls were placed?

Exercise 4.43: Use the `amb` evaluator to solve the following puzzle:[49]

Mary Ann Moore's father has a yacht and so has each of his four friends: Colonel Downing, Mr. Hall, Sir Barnacle Hood, and Dr. Parker. Each of the five also has one daughter and each has named his yacht after a daughter of one of the others. Sir Barnacle's yacht is the Gabrielle, Mr. Moore owns the Lorna; Mr. Hall the Rosalind. The Melissa, owned by Colonel Downing, is named after Sir Barnacle's daughter. Gabrielle's father owns the yacht that is named after Dr. Parker's daughter. Who is Lorna's father?

Try to write the program so that it runs efficiently (see Exercise 4.40). Also determine how many solutions there are if we are not told that Mary Ann's last name is Moore.

Exercise 4.44: Exercise 2.42 described the "eight-queens puzzle" of placing queens on a chessboard so that no two attack each other. Write a nondeterministic program to solve this puzzle.

Parsing natural language

Programs designed to accept natural language as input usually start by attempting to *parse* the input, that is, to match the input against some grammatical structure. For example, we might try to recognize simple

[49] This is taken from a booklet called "Problematical Recreations," published in the 1960s by Litton Industries, where it is attributed to the *Kansas State Engineer*.

sentences consisting of an article followed by a noun followed by a verb, such as "The cat eats." To accomplish such an analysis, we must be able to identify the parts of speech of individual words. We could start with some lists that classify various words:[50]

```
(define nouns '(noun student professor cat class))
(define verbs '(verb studies lectures eats sleeps))
(define articles '(article the a))
```

We also need a *grammar*, that is, a set of rules describing how grammatical elements are composed from simpler elements. A very simple grammar might stipulate that a sentence always consists of two pieces—a noun phrase followed by a verb—and that a noun phrase consists of an article followed by a noun. With this grammar, the sentence "The cat eats" is parsed as follows:

```
(sentence (noun-phrase (article the) (noun cat))
          (verb eats))
```

We can generate such a parse with a simple program that has separate procedures for each of the grammatical rules. To parse a sentence, we identify its two constituent pieces and return a list of these two elements, tagged with the symbol `sentence`:

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-word verbs)))
```

A noun phrase, similarly, is parsed by finding an article followed by a noun:

[50]Here we use the convention that the first element of each list designates the part of speech for the rest of the words in the list.

```
(define (parse-noun-phrase)
  (list 'noun-phrase
        (parse-word articles)
        (parse-word nouns)))
```

At the lowest level, parsing boils down to repeatedly checking that the next unparsed word is a member of the list of words for the required part of speech. To implement this, we maintain a global variable `*unparsed*`, which is the input that has not yet been parsed. Each time we check a word, we require that `*unparsed*` must be non-empty and that it should begin with a word from the designated list. If so, we remove that word from `*unparsed*` and return the word together with its part of speech (which is found at the head of the list):[51]

```
(define (parse-word word-list)
  (require (not (null? *unparsed*)))
  (require (memq (car *unparsed*) (cdr word-list)))
  (let ((found-word (car *unparsed*)))
    (set! *unparsed* (cdr *unparsed*))
    (list (car word-list) found-word)))
```

To start the parsing, all we need to do is set `*unparsed*` to be the entire input, try to parse a sentence, and check that nothing is left over:

```
(define *unparsed* '())
(define (parse input)
  (set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*)) sent))
```

We can now try the parser and verify that it works for our simple test sentence:

[51] Notice that `parse-word` uses `set!` to modify the unparsed input list. For this to work, our `amb` evaluator must undo the effects of `set!` operations when it backtracks.

```
;;; Amb-Eval input:
(parse '(the cat eats))
;;; Starting a new problem
;;; Amb-Eval value:

(sentence (noun-phrase (article the) (noun cat)) (verb eats))
```

The `amb` evaluator is useful here because it is convenient to express the parsing constraints with the aid of `require`. Automatic search and backtracking really pay off, however, when we consider more complex grammars where there are choices for how the units can be decomposed.

Let's add to our grammar a list of prepositions:

```
(define prepositions '(prep for to in by with))
```

and define a prepositional phrase (e.g., "for the cat") to be a preposition followed by a noun phrase:

```
(define (parse-prepositional-phrase)
  (list 'prep-phrase
        (parse-word prepositions)
        (parse-noun-phrase)))
```

Now we can define a sentence to be a noun phrase followed by a verb phrase, where a verb phrase can be either a verb or a verb phrase extended by a prepositional phrase:[52]

```
(define (parse-sentence)
  (list 'sentence (parse-noun-phrase) (parse-verb-phrase)))
(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
```

[52] Observe that this definition is recursive—a verb may be followed by any number of prepositional phrases.

```
        (maybe-extend
         (list 'verb-phrase
               verb-phrase
               (parse-prepositional-phrase)))))
  (maybe-extend (parse-word verbs)))
```

While we're at it, we can also elaborate the definition of noun phrases to permit such things as "a cat in the class." What we used to call a noun phrase, we'll now call a simple noun phrase, and a noun phrase will now be either a simple noun phrase or a noun phrase extended by a prepositional phrase:

```
(define (parse-simple-noun-phrase)
  (list 'simple-noun-phrase
        (parse-word articles)
        (parse-word nouns)))
(define (parse-noun-phrase)
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend
          (list 'noun-phrase
                noun-phrase
                (parse-prepositional-phrase)))))
  (maybe-extend (parse-simple-noun-phrase)))
```

Our new grammar lets us parse more complex sentences. For example

```
(parse '(the student with the cat sleeps in the class))
```

produces

```
(sentence
 (noun-phrase
  (simple-noun-phrase (article the) (noun student))
  (prep-phrase
   (prep with)
```

```
   (simple-noun-phrase (article the) (noun cat))))
 (verb-phrase
  (verb sleeps)
  (prep-phrase
   (prep in)
   (simple-noun-phrase (article the) (noun class)))))
```

Observe that a given input may have more than one legal parse. In the sentence "The professor lectures to the student with the cat," it may be that the professor is lecturing with the cat, or that the student has the cat. Our nondeterministic program finds both possibilities:

```
(parse '(the professor lectures to the student with the cat))
```

produces

```
(sentence
 (simple-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb-phrase
   (verb lectures)
   (prep-phrase
    (prep to)
    (simple-noun-phrase (article the) (noun student))))
  (prep-phrase
   (prep with)
   (simple-noun-phrase (article the) (noun cat)))))
```

Asking the evaluator to try again yields

```
(sentence
 (simple-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb lectures)
  (prep-phrase
   (prep to)
```

```
(noun-phrase
 (simple-noun-phrase (article the) (noun student))
 (prep-phrase
  (prep with)
  (simple-noun-phrase (article the) (noun cat)))))))
```

Exercise 4.45: With the grammar given above, the following sentence can be parsed in five different ways: "The professor lectures to the student in the class with the cat." Give the five parses and explain the differences in shades of meaning among them.

Exercise 4.46: The evaluators in Section 4.1 and Section 4.2 do not determine what order operands are evaluated in. We will see that the `amb` evaluator evaluates them from left to right. Explain why our parsing program wouldn't work if the operands were evaluated in some other order.

Exercise 4.47: Louis Reasoner suggests that, since a verb phrase is either a verb or a verb phrase followed by a prepositional phrase, it would be much more straightforward to define the procedure `parse-verb-phrase` as follows (and similarly for noun phrases):

```
(define (parse-verb-phrase)
  (amb (parse-word verbs)
       (list 'verb-phrase
             (parse-verb-phrase)
             (parse-prepositional-phrase))))
```

Does this work? Does the program's behavior change if we interchange the order of expressions in the `amb`?

Exercise 4.48: Extend the grammar given above to handle more complex sentences. For example, you could extend noun phrases and verb phrases to include adjectives and adverbs, or you could handle compound sentences.[53]

Exercise 4.49: Alyssa P. Hacker is more interested in generating interesting sentences than in parsing them. She reasons that by simply changing the procedure `parse-word` so that it ignores the "input sentence" and instead always succeeds and generates an appropriate word, we can use the programs we had built for parsing to do generation instead. Implement Alyssa's idea, and show the first half-dozen or so sentences generated.[54]

4.3.3 Implementing the `amb` Evaluator

The evaluation of an ordinary Scheme expression may return a value, may never terminate, or may signal an error. In nondeterministic Scheme the evaluation of an expression may in addition result in the discovery of a dead end, in which case evaluation must backtrack to a previous

[53] This kind of grammar can become arbitrarily complex, but it is only a toy as far as real language understanding is concerned. Real natural-language understanding by computer requires an elaborate mixture of syntactic analysis and interpretation of meaning. On the other hand, even toy parsers can be useful in supporting flexible command languages for programs such as information-retrieval systems. Winston 1992 discusses computational approaches to real language understanding and also the applications of simple grammars to command languages.

[54] Although Alyssa's idea works just fine (and is surprisingly simple), the sentences that it generates are a bit boring—they don't sample the possible sentences of this language in a very interesting way. In fact, the grammar is highly recursive in many places, and Alyssa's technique "falls into" one of these recursions and gets stuck. See Exercise 4.50 for a way to deal with this.

choice point. The interpretation of nondeterministic Scheme is complicated by this extra case.

We will construct the `amb` evaluator for nondeterministic Scheme by modifying the analyzing evaluator of Section 4.1.7.[55] As in the analyzing evaluator, evaluation of an expression is accomplished by calling an execution procedure produced by analysis of that expression. The difference between the interpretation of ordinary Scheme and the interpretation of nondeterministic Scheme will be entirely in the execution procedures.

Execution procedures and continuations

Recall that the execution procedures for the ordinary evaluator take one argument: the environment of execution. In contrast, the execution procedures in the `amb` evaluator take three arguments: the environment, and two procedures called *continuation procedures*. The evaluation of an expression will finish by calling one of these two continuations: If the evaluation results in a value, the *success continuation* is called with that value; if the evaluation results in the discovery of a dead end, the *failure continuation* is called. Constructing and calling appropriate continuations is the mechanism by which the nondeterministic evaluator implements backtracking.

It is the job of the success continuation to receive a value and proceed with the computation. Along with that value, the success continuation is passed another failure continuation, which is to be called subsequently if the use of that value leads to a dead end.

[55] We chose to implement the lazy evaluator in Section 4.2 as a modification of the ordinary metacircular evaluator of Section 4.1.1. In contrast, we will base the `amb` evaluator on the analyzing evaluator of Section 4.1.7, because the execution procedures in that evaluator provide a convenient framework for implementing backtracking.

It is the job of the failure continuation to try another branch of the nondeterministic process. The essence of the nondeterministic language is in the fact that expressions may represent choices among alternatives. The evaluation of such an expression must proceed with one of the indicated alternative choices, even though it is not known in advance which choices will lead to acceptable results. To deal with this, the evaluator picks one of the alternatives and passes this value to the success continuation. Together with this value, the evaluator constructs and passes along a failure continuation that can be called later to choose a different alternative.

A failure is triggered during evaluation (that is, a failure continuation is called) when a user program explicitly rejects the current line of attack (for example, a call to `require` may result in execution of `(amb)`, an expression that always fails—see Section 4.3.1). The failure continuation in hand at that point will cause the most recent choice point to choose another alternative. If there are no more alternatives to be considered at that choice point, a failure at an earlier choice point is triggered, and so on. Failure continuations are also invoked by the driver loop in response to a `try-again` request, to find another value of the expression.

In addition, if a side-effect operation (such as assignment to a variable) occurs on a branch of the process resulting from a choice, it may be necessary, when the process finds a dead end, to undo the side effect before making a new choice. This is accomplished by having the side-effect operation produce a failure continuation that undoes the side effect and propagates the failure.

In summary, failure continuations are constructed by

- `amb` expressions—to provide a mechanism to make alternative choices if the current choice made by the `amb` expression leads to a dead end;

- the top-level driver—to provide a mechanism to report failure when the choices are exhausted;
- assignments—to intercept failures and undo assignments during backtracking.

Failures are initiated only when a dead end is encountered. This occurs

- if the user program executes `(amb)`;
- if the user types `try-again` at the top-level driver.

Failure continuations are also called during processing of a failure:

- When the failure continuation created by an assignment finishes undoing a side effect, it calls the failure continuation it intercepted, in order to propagate the failure back to the choice point that led to this assignment or to the top level.
- When the failure continuation for an `amb` runs out of choices, it calls the failure continuation that was originally given to the `amb`, in order to propagate the failure back to the previous choice point or to the top level.

Structure of the evaluator

The syntax- and data-representation procedures for the `amb` evaluator, and also the basic `analyze` procedure, are identical to those in the evaluator of Section 4.1.7, except for the fact that we need additional syntax procedures to recognize the `amb` special form:[56]

[56] We assume that the evaluator supports `let` (see Exercise 4.22), which we have used in our nondeterministic programs.

```
(define (amb? exp) (tagged-list? exp 'amb))
(define (amb-choices exp) (cdr exp))
```

We must also add to the dispatch in `analyze` a clause that will recognize this special form and generate an appropriate execution procedure:

```
((amb? exp) (analyze-amb exp))
```

The top-level procedure `ambeval` (similar to the version of `eval` given in Section 4.1.7) analyzes the given expression and applies the resulting execution procedure to the given environment, together with two given continuations:

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```

A success continuation is a procedure of two arguments: the value just obtained and another failure continuation to be used if that value leads to a subsequent failure. A failure continuation is a procedure of no arguments. So the general form of an execution procedure is

```
(lambda (env succeed fail)
  ;; succeed is (lambda (value fail) ...)
  ;; fail is (lambda () ...)
  ...)
```

For example, executing

```
(ambeval ⟨exp⟩
         the-global-environment
         (lambda (value fail) value)
         (lambda () 'failed))
```

will attempt to evaluate the given expression and will return either the expression's value (if the evaluation succeeds) or the symbol `failed` (if the evaluation fails). The call to `ambeval` in the driver loop shown below

uses much more complicated continuation procedures, which continue the loop and support the `try-again` request.

Most of the complexity of the `amb` evaluator results from the mechanics of passing the continuations around as the execution procedures call each other. In going through the following code, you should compare each of the execution procedures with the corresponding procedure for the ordinary evaluator given in Section 4.1.7.

Simple expressions

The execution procedures for the simplest kinds of expressions are essentially the same as those for the ordinary evaluator, except for the need to manage the continuations. The execution procedures simply succeed with the value of the expression, passing along the failure continuation that was passed to them.

```
(define (analyze-self-evaluating exp)
  (lambda (env succeed fail)
    (succeed exp fail)))
(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env succeed fail)
      (succeed qval fail))))
(define (analyze-variable exp)
  (lambda (env succeed fail)
    (succeed (lookup-variable-value exp env) fail)))
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env succeed fail)
      (succeed (make-procedure vars bproc env) fail))))
```

Notice that looking up a variable always 'succeeds.' If `lookup-variable-value` fails to find the variable, it signals an error, as usual. Such a "failure" indicates a program bug—a reference to an unbound variable; it is not an indication that we should try another nondeterministic choice instead of the one that is currently being tried.

Conditionals and sequences

Conditionals are also handled in a similar way as in the ordinary evaluator. The execution procedure generated by `analyze-if` invokes the predicate execution procedure `pproc` with a success continuation that checks whether the predicate value is true and goes on to execute either the consequent or the alternative. If the execution of `pproc` fails, the original failure continuation for the `if` expression is called.

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env
             ;; success continuation for evaluating the predicate
             ;; to obtain pred-value
             (lambda (pred-value fail2)
               (if (true? pred-value)
                   (cproc env succeed fail2)
                   (aproc env succeed fail2)))
             ;; failure continuation for evaluating the predicate
             fail))))
```

Sequences are also handled in the same way as in the previous evaluator, except for the machinations in the subprocedure `sequentially` that are required for passing the continuations. Namely, to sequentially execute

a and then b, we call a with a success continuation that calls b.

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
         ;; success continuation for calling a
         (lambda (a-value fail2)
           (b env succeed fail2))
         ;; failure continuation for calling a
         fail)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc
                            (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence: ANALYZE"))
    (loop (car procs) (cdr procs))))
```

Definitions and assignments

Definitions are another case where we must go to some trouble to manage the continuations, because it is necessary to evaluate the definition-value expression before actually defining the new variable. To accomplish this, the definition-value execution procedure `vproc` is called with the environment, a success continuation, and the failure continuation. If the execution of `vproc` succeeds, obtaining a value `val` for the defined variable, the variable is defined and the success is propagated:

```
(define (analyze-definition exp)
```

```
(let ((var (definition-variable exp))
      (vproc (analyze (definition-value exp))))
  (lambda (env succeed fail)
    (vproc env
           (lambda (val fail2)
             (define-variable! var val env)
             (succeed 'ok fail2))
           fail))))
```

Assignments are more interesting. This is the first place where we really use the continuations, rather than just passing them around. The execution procedure for assignments starts out like the one for definitions. It first attempts to obtain the new value to be assigned to the variable. If this evaluation of `vproc` fails, the assignment fails.

If `vproc` succeeds, however, and we go on to make the assignment, we must consider the possibility that this branch of the computation might later fail, which will require us to backtrack out of the assignment. Thus, we must arrange to undo the assignment as part of the backtracking process.[57]

This is accomplished by giving `vproc` a success continuation (marked with the comment "*1*" below) that saves the old value of the variable before assigning the new value to the variable and proceeding from the assignment. The failure continuation that is passed along with the value of the assignment (marked with the comment "*2*" below) restores the old value of the variable before continuing the failure. That is, a successful assignment provides a failure continuation that will intercept a subsequent failure; whatever failure would otherwise have called `fail2` calls this procedure instead, to undo the assignment before actually calling `fail2`.

[57] We didn't worry about undoing definitions, since we can assume that internal definitions are scanned out (Section 4.1.6).

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail2)        ;*1*
               (let ((old-value
                      (lookup-variable-value var env)))
                 (set-variable-value! var val env)
                 (succeed 'ok
                          (lambda ()    ;*2*
                            (set-variable-value!
                             var old-value env)
                            (fail2)))))
             fail))))
```

Procedure applications

The execution procedure for applications contains no new ideas except for the technical complexity of managing the continuations. This complexity arises in `analyze-application`, due to the need to keep track of the success and failure continuations as we evaluate the operands. We use a procedure `get-args` to evaluate the list of operands, rather than a simple `map` as in the ordinary evaluator.

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
             (lambda (proc fail2)
               (get-args aprocs
                         env
```

```
                    (lambda (args fail3)
                      (execute-application
                       proc args succeed fail3))
                    fail2))
          fail))))
```

In `get-args`, notice how `cdr`-ing down the list of `aproc` execution procedures and `cons`ing up the resulting list of `args` is accomplished by calling each `aproc` in the list with a success continuation that recursively calls `get-args`. Each of these recursive calls to `get-args` has a success continuation whose value is the `cons` of the newly obtained argument onto the list of accumulated arguments:

```
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs)
       env
       ;; success continuation for this aproc
       (lambda (arg fail2)
         (get-args
          (cdr aprocs)
          env
          ;; success continuation for
          ;; recursive call to get-args
          (lambda (args fail3)
            (succeed (cons arg args) fail3))
          fail2))
       fail)))
```

The actual procedure application, which is performed by `execute-application`, is accomplished in the same way as for the ordinary evaluator, except for the need to manage the continuations.

```
(define (execute-application proc args succeed fail)
  (cond ((primitive-procedure? proc)
         (succeed (apply-primitive-procedure proc args)
                  fail))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment
           (procedure-parameters proc)
           args
           (procedure-environment proc))
          succeed
          fail))
        (else
         (error "Unknown procedure type: EXECUTE-APPLICATION"
                proc))))
```

Evaluating `amb` expressions

The `amb` special form is the key element in the nondeterministic language. Here we see the essence of the interpretation process and the reason for keeping track of the continuations. The execution procedure for `amb` defines a loop `try-next` that cycles through the execution procedures for all the possible values of the `amb` expression. Each execution procedure is called with a failure continuation that will try the next one. When there are no more alternatives to try, the entire `amb` expression fails.

```
(define (analyze-amb exp)
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
```

```
          ((car choices)
           env
           succeed
           (lambda () (try-next (cdr choices))))))
    (try-next cprocs))))
```

Driver loop

The driver loop for the `amb` evaluator is complex, due to the mechanism that permits the user to try again in evaluating an expression. The driver uses a procedure called `internal-loop`, which takes as argument a procedure `try-again`. The intent is that calling `try-again` should go on to the next untried alternative in the nondeterministic evaluation. `internal-loop` either calls `try-again` in response to the user typing `try-again` at the driver loop, or else starts a new evaluation by calling `ambeval`.

The failure continuation for this call to `ambeval` informs the user that there are no more values and re-invokes the driver loop.

The success continuation for the call to `ambeval` is more subtle. We print the obtained value and then invoke the internal loop again with a `try-again` procedure that will be able to try the next alternative. This `next-alternative` procedure is the second argument that was passed to the success continuation. Ordinarily, we think of this second argument as a failure continuation to be used if the current evaluation branch later fails. In this case, however, we have completed a successful evaluation, so we can invoke the "failure" alternative branch in order to search for additional successful evaluations.

```
(define input-prompt  ";;; Amb-Eval input:")
(define output-prompt ";;; Amb-Eval value:")
```

```
(define (driver-loop)
  (define (internal-loop try-again)
    (prompt-for-input input-prompt)
    (let ((input (read)))
      (if (eq? input 'try-again)
          (try-again)
          (begin
            (newline) (display ";;; Starting a new problem ")
            (ambeval
             input
             the-global-environment
             ;; ambeval success
             (lambda (val next-alternative)
               (announce-output output-prompt)
               (user-print val)
               (internal-loop next-alternative))
             ;; ambeval failure
             (lambda ()
               (announce-output
                ";;; There are no more values of")
               (user-print input)
               (driver-loop)))))))
  (internal-loop
   (lambda ()
     (newline) (display ";;; There is no current problem")
     (driver-loop))))
```

The initial call to `internal-loop` uses a `try-again` procedure that complains that there is no current problem and restarts the driver loop. This is the behavior that will happen if the user types `try-again` when there is no evaluation in progress.

Exercise 4.50: Implement a new special form `ramb` that is

like amb except that it searches alternatives in a random order, rather than from left to right. Show how this can help with Alyssa's problem in Exercise 4.49.

Exercise 4.51: Implement a new kind of assignment called permanent-set! that is not undone upon failure. For example, we can choose two distinct elements from a list and count the number of trials required to make a successful choice as follows:

```
(define count 0)
(let ((x (an-element-of '(a b c)))
      (y (an-element-of '(a b c))))
  (permanent-set! count (+ count 1))
  (require (not (eq? x y)))
  (list x y count))
;;; Starting a new problem
;;; Amb-Eval value:
(a b 2)
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
(a c 3)
```

What values would have been displayed if we had used set! here rather than permanent-set! ?

Exercise 4.52: Implement a new construct called if-fail that permits the user to catch the failure of an expression. if-fail takes two expressions. It evaluates the first expression as usual and returns as usual if the evaluation succeeds. If the evaluation fails, however, the value of the second expression is returned, as in the following example:

```
;;; Amb-Eval input:
(if-fail (let ((x (an-element-of '(1 3 5))))
           (require (even? x))
           x)
         'all-odd)
;;; Starting a new problem
;;; Amb-Eval value:
all-odd

;;; Amb-Eval input:
(if-fail (let ((x (an-element-of '(1 3 5 8))))
           (require (even? x))
           x)
         'all-odd)
;;; Starting a new problem
;;; Amb-Eval value:
8
```

Exercise 4.53: With `permanent-set!` as described in Exercise 4.51 and `if-fail` as in Exercise 4.52, what will be the result of evaluating

```
(let ((pairs '()))
  (if-fail
   (let ((p (prime-sum-pair '(1 3 5 8)
                            '(20 35 110))))
     (permanent-set! pairs (cons p pairs))
     (amb))
   pairs))
```

Exercise 4.54: If we had not realized that `require` could be implemented as an ordinary procedure that uses `amb`, to be defined by the user as part of a nondeterministic program,

we would have had to implement it as a special form. This would require syntax procedures

```
(define (require? exp)
  (tagged-list? exp 'require))
(define (require-predicate exp)
  (cadr exp))
```

and a new clause in the dispatch in `analyze`

```
((require? exp) (analyze-require exp))
```

as well the procedure `analyze-require` that handles `require` expressions. Complete the following definition of `analyze-require`.

```
(define (analyze-require exp)
  (let ((pproc (analyze (require-predicate exp))))
    (lambda (env succeed fail)
      (pproc env
             (lambda (pred-value fail2)
               (if ⟨??⟩
                   ⟨??⟩
                   (succeed 'ok fail2)))
             fail))))
```

4.4 Logic Programming

In Chapter 1 we stressed that computer science deals with imperative (how to) knowledge, whereas mathematics deals with declarative (what is) knowledge. Indeed, programming languages require that the programmer express knowledge in a form that indicates the step-by-step methods for solving particular problems. On the other hand, high-level

languages provide, as part of the language implementation, a substantial amount of methodological knowledge that frees the user from concern with numerous details of how a specified computation will progress.

Most programming languages, including Lisp, are organized around computing the values of mathematical functions. Expression-oriented languages (such as Lisp, Fortran, and Algol) capitalize on the "pun" that an expression that describes the value of a function may also be interpreted as a means of computing that value. Because of this, most programming languages are strongly biased toward unidirectional computations (computations with well-defined inputs and outputs). There are, however, radically different programming languages that relax this bias. We saw one such example in Section 3.3.5, where the objects of computation were arithmetic constraints. In a constraint system the direction and the order of computation are not so well specified; in carrying out a computation the system must therefore provide more detailed "how to" knowledge than would be the case with an ordinary arithmetic computation. This does not mean, however, that the user is released altogether from the responsibility of providing imperative knowledge. There are many constraint networks that implement the same set of constraints, and the user must choose from the set of mathematically equivalent networks a suitable network to specify a particular computation.

The nondeterministic program evaluator of Section 4.3 also moves away from the view that programming is about constructing algorithms for computing unidirectional functions. In a nondeterministic language, expressions can have more than one value, and, as a result, the computation is dealing with relations rather than with single-valued functions. Logic programming extends this idea by combining a relational vision of programming with a powerful kind of symbolic pattern matching

called *unification*.[58]

This approach, when it works, can be a very powerful way to write programs. Part of the power comes from the fact that a single "what is" fact can be used to solve a number of different problems that would have different "how to" components. As an example, consider the append operation, which takes two lists as arguments and combines their elements to form a single list. In a procedural language such as Lisp, we could define append in terms of the basic list constructor cons, as we did in Section 2.2.1:

[58]Logic programming has grown out of a long history of research in automatic theorem proving. Early theorem-proving programs could accomplish very little, because they exhaustively searched the space of possible proofs. The major breakthrough that made such a search plausible was the discovery in the early 1960s of the *unification algorithm* and the *resolution principle* (Robinson 1965). Resolution was used, for example, by Green and Raphael (1968) (see also Green 1969) as the basis for a deductive question-answering system. During most of this period, researchers concentrated on algorithms that are guaranteed to find a proof if one exists. Such algorithms were difficult to control and to direct toward a proof. Hewitt (1969) recognized the possibility of merging the control structure of a programming language with the operations of a logic-manipulation system, leading to the work in automatic search mentioned in Section 4.3.1 (Footnote 4.47). At the same time that this was being done, Colmerauer, in Marseille, was developing rule-based systems for manipulating natural language (see Colmerauer et al. 1973). He invented a programming language called Prolog for representing those rules. Kowalski (1973; 1979), in Edinburgh, recognized that execution of a Prolog program could be interpreted as proving theorems (using a proof technique called linear Horn-clause resolution). The merging of the last two strands led to the logic-programming movement. Thus, in assigning credit for the development of logic programming, the French can point to Prolog's genesis at the University of Marseille, while the British can highlight the work at the University of Edinburgh. According to people at MIT, logic programming was developed by these groups in an attempt to figure out what Hewitt was talking about in his brilliant but impenetrable Ph.D. thesis. For a history of logic programming, see Robinson 1983.

```
(define (append x y)
  (if (null? x) y (cons (car x) (append (cdr x) y))))
```

This procedure can be regarded as a translation into Lisp of the following two rules, the first of which covers the case where the first list is empty and the second of which handles the case of a nonempty list, which is a cons of two parts:

- For any list y, the empty list and y append to form y.
- For any u, v, y, and z, (cons u v) and y append to form (cons u z) if v and y append to form z.[59]

Using the append procedure, we can answer questions such as

> Find the append of (a b) and (c d).

But the same two rules are also sufficient for answering the following sorts of questions, which the procedure can't answer:

> Find a list y that appends with (a b) to produce (a b c d).
>
> Find all x and y that append to form (a b c d).

In a logic programming language, the programmer writes an append "procedure" by stating the two rules about append given above. "How to" knowledge is provided automatically by the interpreter to allow this

[59] To see the correspondence between the rules and the procedure, let x in the procedure (where x is nonempty) correspond to (cons u v) in the rule. Then z in the rule corresponds to the append of (cdr x) and y.

single pair of rules to be used to answer all three types of questions about append.[60]

Contemporary logic programming languages (including the one we implement here) have substantial deficiencies, in that their general "how to" methods can lead them into spurious infinite loops or other undesirable behavior. Logic programming is an active field of research in computer science.[61]

Earlier in this chapter we explored the technology of implementing interpreters and described the elements that are essential to an interpreter for a Lisp-like language (indeed, to an interpreter for any conventional language). Now we will apply these ideas to discuss an interpreter for a logic programming language. We call this language the *query language*, because it is very useful for retrieving information from data bases by formulating *queries*, or questions, expressed in the language. Even though the query language is very different from Lisp, we

[60] This certainly does not relieve the user of the entire problem of how to compute the answer. There are many different mathematically equivalent sets of rules for formulating the append relation, only some of which can be turned into effective devices for computing in any direction. In addition, sometimes "what is" information gives no clue "how to" compute an answer. For example, consider the problem of computing the y such that $y^2 = x$.

[61] Interest in logic programming peaked during the early 80s when the Japanese government began an ambitious project aimed at building superfast computers optimized to run logic programming languages. The speed of such computers was to be measured in LIPS (Logical Inferences Per Second) rather than the usual FLOPS (FLoating-point Operations Per Second). Although the project succeeded in developing hardware and software as originally planned, the international computer industry moved in a different direction. See Feigenbaum and Shrobe 1993 for an overview evaluation of the Japanese project. The logic programming community has also moved on to consider relational programming based on techniques other than simple pattern matching, such as the ability to deal with numerical constraints such as the ones illustrated in the constraint-propagation system of Section 3.3.5.

will find it convenient to describe the language in terms of the same general framework we have been using all along: as a collection of primitive elements, together with means of combination that enable us to combine simple elements to create more complex elements and means of abstraction that enable us to regard complex elements as single conceptual units. An interpreter for a logic programming language is considerably more complex than an interpreter for a language like Lisp. Nevertheless, we will see that our query-language interpreter contains many of the same elements found in the interpreter of Section 4.1. In particular, there will be an "eval" part that classifies expressions according to type and an "apply" part that implements the language's abstraction mechanism (procedures in the case of Lisp, and *rules* in the case of logic programming). Also, a central role is played in the implementation by a frame data structure, which determines the correspondence between symbols and their associated values. One additional interesting aspect of our query-language implementation is that we make substantial use of streams, which were introduced in Chapter 3.

4.4.1 Deductive Information Retrieval

Logic programming excels in providing interfaces to data bases for information retrieval. The query language we shall implement in this chapter is designed to be used in this way.

In order to illustrate what the query system does, we will show how it can be used to manage the data base of personnel records for Microshaft, a thriving high-technology company in the Boston area. The language provides pattern-directed access to personnel information and can also take advantage of general rules in order to make logical deductions.

A sample data base

The personnel data base for Microshaft contains *assertions* about company personnel. Here is the information about Ben Bitdiddle, the resident computer wizard:

```
(address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))
(job (Bitdiddle Ben) (computer wizard))
(salary (Bitdiddle Ben) 60000)
```

Each assertion is a list (in this case a triple) whose elements can themselves be lists.

As resident wizard, Ben is in charge of the company's computer division, and he supervises two programmers and one technician. Here is the information about them:

```
(address (Hacker Alyssa P) (Cambridge (Mass Ave) 78))
(job (Hacker Alyssa P) (computer programmer))
(salary (Hacker Alyssa P) 40000)
(supervisor (Hacker Alyssa P) (Bitdiddle Ben))

(address (Fect Cy D) (Cambridge (Ames Street) 3))
(job (Fect Cy D) (computer programmer))
(salary (Fect Cy D) 35000)
(supervisor (Fect Cy D) (Bitdiddle Ben))

(address (Tweakit Lem E) (Boston (Bay State Road) 22))
(job (Tweakit Lem E) (computer technician))
(salary (Tweakit Lem E) 25000)
(supervisor (Tweakit Lem E) (Bitdiddle Ben))
```

There is also a programmer trainee, who is supervised by Alyssa:

```
(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(job (Reasoner Louis) (computer programmer trainee))
```

```
(salary (Reasoner Louis) 30000)
(supervisor (Reasoner Louis) (Hacker Alyssa P))
```

All of these people are in the computer division, as indicated by the word `computer` as the first item in their job descriptions.

Ben is a high-level employee. His supervisor is the company's big wheel himself:

```
(supervisor (Bitdiddle Ben) (Warbucks Oliver))
(address (Warbucks Oliver) (Swellesley (Top Heap Road)))
(job (Warbucks Oliver) (administration big wheel))
(salary (Warbucks Oliver) 150000)
```

Besides the computer division supervised by Ben, the company has an accounting division, consisting of a chief accountant and his assistant:

```
(address (Scrooge Eben) (Weston (Shady Lane) 10))
(job (Scrooge Eben) (accounting chief accountant))
(salary (Scrooge Eben) 75000)
(supervisor (Scrooge Eben) (Warbucks Oliver))

(address (Cratchet Robert) (Allston (N Harvard Street) 16))
(job (Cratchet Robert) (accounting scrivener))
(salary (Cratchet Robert) 18000)
(supervisor (Cratchet Robert) (Scrooge Eben))
```

There is also a secretary for the big wheel:

```
(address (Aull DeWitt) (Slumerville (Onion Square) 5))
(job (Aull DeWitt) (administration secretary))
(salary (Aull DeWitt) 25000)
(supervisor (Aull DeWitt) (Warbucks Oliver))
```

The data base also contains assertions about which kinds of jobs can be done by people holding other kinds of jobs. For instance, a computer

wizard can do the jobs of both a computer programmer and a computer technician:

```
(can-do-job (computer wizard) (computer programmer))
(can-do-job (computer wizard) (computer technician))
```

A computer programmer could fill in for a trainee:

```
(can-do-job (computer programmer)
            (computer programmer trainee))
```

Also, as is well known,

```
(can-do-job (administration secretary)
            (administration big wheel))
```

Simple queries

The query language allows users to retrieve information from the data base by posing queries in response to the system's prompt. For example, to find all computer programmers one can say

```
;;; Query input:
(job ?x (computer programmer))
```

The system will respond with the following items:

```
;;; Query results:
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (computer programmer))
```

The input query specifies that we are looking for entries in the data base that match a certain *pattern*. In this example, the pattern specifies entries consisting of three items, of which the first is the literal symbol `job`, the second can be anything, and the third is the literal list `(computer programmer)`. The "anything" that can be the second item in the matching list is specified by a *pattern variable*, `?x`. The general form of a pattern

variable is a symbol, taken to be the name of the variable, preceded by a question mark. We will see below why it is useful to specify names for pattern variables rather than just putting ? into patterns to represent "anything." The system responds to a simple query by showing all entries in the data base that match the specified pattern.

A pattern can have more than one variable. For example, the query

```
(address ?x ?y)
```

will list all the employees' addresses.

A pattern can have no variables, in which case the query simply determines whether that pattern is an entry in the data base. If so, there will be one match; if not, there will be no matches.

The same pattern variable can appear more than once in a query, specifying that the same "anything" must appear in each position. This is why variables have names. For example,

```
(supervisor ?x ?x)
```

finds all people who supervise themselves (though there are no such assertions in our sample data base).

The query

```
(job ?x (computer ?type))
```

matches all job entries whose third item is a two-element list whose first item is `computer`:

```
(job (Bitdiddle Ben) (computer wizard))
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (computer programmer))
(job (Tweakit Lem E) (computer technician))
```

This same pattern does *not* match

```
(job (Reasoner Louis) (computer programmer trainee))
```

because the third item in the entry is a list of three elements, and the pattern's third item specifies that there should be two elements. If we wanted to change the pattern so that the third item could be any list beginning with `computer`, we could specify[62]

```
(job ?x (computer . ?type))
```

For example,

```
(computer . ?type)
```

matches the data

```
(computer programmer trainee)
```

with `?type` as the list `(programmer trainee)`. It also matches the data

```
(computer programmer)
```

with `?type` as the list `(programmer)`, and matches the data

```
(computer)
```

with `?type` as the empty list `()`.

We can describe the query language's processing of simple queries as follows:

- The system finds all assignments to variables in the query pattern that *satisfy* the pattern—that is, all sets of values for the variables such that if the pattern variables are *instantiated with* (replaced by) the values, the result is in the data base.
- The system responds to the query by listing all instantiations of the query pattern with the variable assignments that satisfy it.

[62] This uses the dotted-tail notation introduced in Exercise 2.20.

Note that if the pattern has no variables, the query reduces to a determination of whether that pattern is in the data base. If so, the empty assignment, which assigns no values to variables, satisfies that pattern for that data base.

> **Exercise 4.55:** Give simple queries that retrieve the following information from the data base:
>
> 1. all people supervised by Ben Bitdiddle;
> 2. the names and jobs of all people in the accounting division;
> 3. the names and addresses of all people who live in Slumerville.

Compound queries

Simple queries form the primitive operations of the query language. In order to form compound operations, the query language provides means of combination. One thing that makes the query language a logic programming language is that the means of combination mirror the means of combination used in forming logical expressions: `and`, `or`, and `not`. (Here `and`, `or`, and `not` are not the Lisp primitives, but rather operations built into the query language.)

We can use `and` as follows to find the addresses of all the computer programmers:

```
(and (job ?person (computer programmer))
     (address ?person ?where))
```

The resulting output is

```
(and (job (Hacker Alyssa P) (computer programmer))
     (address (Hacker Alyssa P) (Cambridge (Mass Ave) 78)))
```

```
(and (job (Fect Cy D) (computer programmer))
     (address (Fect Cy D) (Cambridge (Ames Street) 3)))
```

In general,

```
(and ⟨query1⟩ ⟨query2⟩ ... ⟨queryn⟩)
```

is satisfied by all sets of values for the pattern variables that simultaneously satisfy $\langle query_1 \rangle \ldots \langle query_n \rangle$.

As for simple queries, the system processes a compound query by finding all assignments to the pattern variables that satisfy the query, then displaying instantiations of the query with those values.

Another means of constructing compound queries is through `or`. For example,

```
(or (supervisor ?x (Bitdiddle Ben))
    (supervisor ?x (Hacker Alyssa P)))
```

will find all employees supervised by Ben Bitdiddle or Alyssa P. Hacker:

```
(or (supervisor (Hacker Alyssa P) (Bitdiddle Ben))
    (supervisor (Hacker Alyssa P) (Hacker Alyssa P)))
(or (supervisor (Fect Cy D) (Bitdiddle Ben))
    (supervisor (Fect Cy D) (Hacker Alyssa P)))
(or (supervisor (Tweakit Lem E) (Bitdiddle Ben))
    (supervisor (Tweakit Lem E) (Hacker Alyssa P)))
(or (supervisor (Reasoner Louis) (Bitdiddle Ben))
    (supervisor (Reasoner Louis) (Hacker Alyssa P)))
```

In general,

```
(or ⟨query1⟩ ⟨query2⟩ ... ⟨queryn⟩)
```

is satisfied by all sets of values for the pattern variables that satisfy at least one of $\langle query_1 \rangle \ldots \langle query_n \rangle$.

Compound queries can also be formed with `not`. For example,

```
(and (supervisor ?x (Bitdiddle Ben))
     (not (job ?x (computer programmer))))
```

finds all people supervised by Ben Bitdiddle who are not computer programmers. In general,

```
(not ⟨query1⟩)
```

is satisfied by all assignments to the pattern variables that do not satisfy ⟨*query*$_1$⟩.[63]

The final combining form is called `lisp-value`. When `lisp-value` is the first element of a pattern, it specifies that the next element is a Lisp predicate to be applied to the rest of the (instantiated) elements as arguments. In general,

```
(lisp-value ⟨predicate⟩ ⟨arg1⟩ ... ⟨argn⟩)
```

will be satisfied by assignments to the pattern variables for which the ⟨*predicate*⟩ applied to the instantiated ⟨*arg*$_1$⟩ . . . ⟨*arg*$_n$⟩ is true. For example, to find all people whose salary is greater than $30,000 we could write[64]

```
(and (salary ?person ?amount) (lisp-value > ?amount 30000))
```

Exercise 4.56: Formulate compound queries that retrieve the following information:

[63] Actually, this description of `not` is valid only for simple cases. The real behavior of `not` is more complex. We will examine `not`'s peculiarities in sections Section 4.4.2 and Section 4.4.3.

[64] `lisp-value` should be used only to perform an operation not provided in the query language. In particular, it should not be used to test equality (since that is what the matching in the query language is designed to do) or inequality (since that can be done with the `same` rule shown below).

a. the names of all people who are supervised by Ben Bitdiddle, together with their addresses;

b. all people whose salary is less than Ben Bitdiddle's, together with their salary and Ben Bitdiddle's salary;

c. all people who are supervised by someone who is not in the computer division, together with the supervisor's name and job.

Rules

In addition to primitive queries and compound queries, the query language provides means for abstracting queries. These are given by *rules*. The rule

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
           (address ?person-2 (?town . ?rest-2))
           (not (same ?person-1 ?person-2))))
```

specifies that two people live near each other if they live in the same town. The final `not` clause prevents the rule from saying that all people live near themselves. The `same` relation is defined by a very simple rule:[65]

```
(rule (same ?x ?x))
```

[65] Notice that we do not need `same` in order to make two things be the same: We just use the same pattern variable for each—in effect, we have one thing instead of two things in the first place. For example, see `?town` in the `lives-near` rule and `?middle-manager` in the `wheel` rule below. `same` is useful when we want to force two things to be different, such as `?person-1` and `?person-2` in the `lives-near` rule. Although using the same pattern variable in two parts of a query forces the same value to appear in both places, using different pattern variables does not force different values to appear. (The values assigned to different pattern variables may be the same or different.)

The following rule declares that a person is a "wheel" in an organization if he supervises someone who is in turn a supervisor:

```
(rule (wheel ?person)
      (and (supervisor ?middle-manager ?person)
           (supervisor ?x ?middle-manager)))
```

The general form of a rule is

```
(rule ⟨conclusion⟩ ⟨body⟩)
```

where ⟨*conclusion*⟩ is a pattern and ⟨*body*⟩ is any query.[66] We can think of a rule as representing a large (even infinite) set of assertions, namely all instantiations of the rule conclusion with variable assignments that satisfy the rule body. When we described simple queries (patterns), we said that an assignment to variables satisfies a pattern if the instantiated pattern is in the data base. But the pattern needn't be explicitly in the data base as an assertion. It can be an implicit assertion implied by a rule. For example, the query

```
(lives-near ?x (Bitdiddle Ben))
```

results in

```
(lives-near (Reasoner Louis) (Bitdiddle Ben))
(lives-near (Aull DeWitt) (Bitdiddle Ben))
```

To find all computer programmers who live near Ben Bitdiddle, we can ask

```
(and (job ?x (computer programmer))
     (lives-near ?x (Bitdiddle Ben)))
```

[66] We will also allow rules without bodies, as in `same`, and we will interpret such a rule to mean that the rule conclusion is satisfied by any values of the variables.

As in the case of compound procedures, rules can be used as parts of other rules (as we saw with the `lives-near` rule above) or even be defined recursively. For instance, the rule

```
(rule (outranked-by ?staff-person ?boss)
      (or (supervisor ?staff-person ?boss)
          (and (supervisor ?staff-person ?middle-manager)
               (outranked-by ?middle-manager ?boss))))
```

says that a staff person is outranked by a boss in the organization if the boss is the person's supervisor or (recursively) if the person's supervisor is outranked by the boss.

Exercise 4.57: Define a rule that says that person 1 can replace person 2 if either person 1 does the same job as person 2 or someone who does person 1's job can also do person 2's job, and if person 1 and person 2 are not the same person. Using your rule, give queries that find the following:

a. all people who can replace Cy D. Fect;

b. all people who can replace someone who is being paid more than they are, together with the two salaries.

Exercise 4.58: Define a rule that says that a person is a "big shot" in a division if the person works in the division but does not have a supervisor who works in the division.

Exercise 4.59: Ben Bitdiddle has missed one meeting too many. Fearing that his habit of forgetting meetings could cost him his job, Ben decides to do something about it. He adds all the weekly meetings of the firm to the Microshaft data base by asserting the following:

```
(meeting accounting (Monday 9am))
(meeting administration (Monday 10am))
(meeting computer (Wednesday 3pm))
(meeting administration (Friday 1pm))
```

Each of the above assertions is for a meeting of an entire division. Ben also adds an entry for the company-wide meeting that spans all the divisions. All of the company's employees attend this meeting.

```
(meeting whole-company (Wednesday 4pm))
```

a. On Friday morning, Ben wants to query the data base for all the meetings that occur that day. What query should he use?

b. Alyssa P. Hacker is unimpressed. She thinks it would be much more useful to be able to ask for her meetings by specifying her name. So she designs a rule that says that a person's meetings include all `whole-company` meetings plus all meetings of that person's division. Fill in the body of Alyssa's rule.

```
(rule (meeting-time ?person ?day-and-time)
      ⟨rule-body⟩)
```

c. Alyssa arrives at work on Wednesday morning and wonders what meetings she has to attend that day. Having defined the above rule, what query should she make to find this out?

Exercise 4.60: By giving the query

```
(lives-near ?person (Hacker Alyssa P))
```

Alyssa P. Hacker is able to find people who live near her, with whom she can ride to work. On the other hand, when she tries to find all pairs of people who live near each other by querying

```
(lives-near ?person-1 ?person-2)
```

she notices that each pair of people who live near each other is listed twice; for example,

```
(lives-near (Hacker Alyssa P) (Fect Cy D))
(lives-near (Fect Cy D) (Hacker Alyssa P))
```

Why does this happen? Is there a way to find a list of people who live near each other, in which each pair appears only once? Explain.

Logic as programs

We can regard a rule as a kind of logical implication: *If* an assignment of values to pattern variables satisfies the body, *then* it satisfies the conclusion. Consequently, we can regard the query language as having the ability to perform *logical deductions* based upon the rules. As an example, consider the `append` operation described at the beginning of Section 4.4. As we said, `append` can be characterized by the following two rules:

- For any `list` `y`, the empty list and `y` `append` to form `y`.
- For any `u`, `v`, `y`, and `z`, `(cons u v)` and `y` `append` to form `(cons u z)` if `v` and `y` `append` to form `z`.

To express this in our query language, we define two rules for a relation

```
(append-to-form x y z)
```

which we can interpret to mean "x and y append to form z":

```
(rule (append-to-form () ?y ?y))
(rule (append-to-form (?u . ?v) ?y (?u . ?z))
      (append-to-form ?v ?y ?z))
```

The first rule has no body, which means that the conclusion holds for any value of `?y`. Note how the second rule makes use of dotted-tail notation to name the `car` and `cdr` of a list.

Given these two rules, we can formulate queries that compute the append of two lists:

```
;;; Query input:
(append-to-form (a b) (c d) ?z)
;;; Query results:
(append-to-form (a b) (c d) (a b c d))
```

What is more striking, we can use the same rules to ask the question "Which list, when appended to `(a b)`, yields `(a b c d)`?" This is done as follows:

```
;;; Query input:
(append-to-form (a b) ?y (a b c d))
;;; Query results:
(append-to-form (a b) (c d) (a b c d))
```

We can also ask for all pairs of lists that append to form `(a b c d)`:

```
;;; Query input:
(append-to-form ?x ?y (a b c d))
;;; Query results:
(append-to-form () (a b c d) (a b c d))
(append-to-form (a) (b c d) (a b c d))
(append-to-form (a b) (c d) (a b c d))
(append-to-form (a b c) (d) (a b c d))
(append-to-form (a b c d) () (a b c d))
```

The query system may seem to exhibit quite a bit of intelligence in using the rules to deduce the answers to the queries above. Actually, as we will see in the next section, the system is following a well-determined algorithm in unraveling the rules. Unfortunately, although the system works impressively in the append case, the general methods may break down in more complex cases, as we will see in Section 4.4.3.

Exercise 4.61: The following rules implement a `next-to` relation that finds adjacent elements of a list:

```
(rule (?x next-to ?y in (?x ?y . ?u)))
(rule (?x next-to ?y in (?v . ?z))
      (?x next-to ?y in ?z))
```

What will the response be to the following queries?

```
(?x next-to ?y in (1 (2 3) 4))
(?x next-to  1 in (2 1 3 1))
```

Exercise 4.62: Define rules to implement the `last-pair` operation of Exercise 2.17, which returns a list containing the last element of a nonempty list. Check your rules on queries such as `(last-pair (3) ?x)`, `(last-pair (1 2 3) ?x)` and `(last-pair (2 ?x) (3))`. Do your rules work correctly on queries such as `(last-pair ?x (3))` ?

Exercise 4.63: The following data base (see Genesis 4) traces the genealogy of the descendants of Ada back to Adam, by way of Cain:

```
(son Adam Cain)
(son Cain Enoch)
(son Enoch Irad)
```

```
(son Irad Mehujael)
(son Mehujael Methushael)
(son Methushael Lamech)
(wife Lamech Ada)
(son Ada Jabal)
(son Ada Jubal)
```

Formulate rules such as "If S is the son of f, and f is the son of G, then S is the grandson of G" and "If W is the wife of M, and S is the son of W, then S is the son of M" (which was supposedly more true in biblical times than today) that will enable the query system to find the grandson of Cain; the sons of Lamech; the grandsons of Methushael. (See Exercise 4.69 for some rules to deduce more complicated relationships.)

4.4.2 How the Query System Works

In Section 4.4.4 we will present an implementation of the query interpreter as a collection of procedures. In this section we give an overview that explains the general structure of the system independent of low-level implementation details. After describing the implementation of the interpreter, we will be in a position to understand some of its limitations and some of the subtle ways in which the query language's logical operations differ from the operations of mathematical logic.

It should be apparent that the query evaluator must perform some kind of search in order to match queries against facts and rules in the data base. One way to do this would be to implement the query system as a nondeterministic program, using the `amb` evaluator of Section 4.3 (see Exercise 4.78). Another possibility is to manage the search with the aid of streams. Our implementation follows this second approach.

The query system is organized around two central operations called *pattern matching* and *unification*. We first describe pattern matching and explain how this operation, together with the organization of information in terms of streams of frames, enables us to implement both simple and compound queries. We next discuss unification, a generalization of pattern matching needed to implement rules. Finally, we show how the entire query interpreter fits together through a procedure that classifies expressions in a manner analogous to the way `eval` classifies expressions for the interpreter described in Section 4.1.

Pattern matching

A *pattern matcher* is a program that tests whether some datum fits a specified pattern. For example, the data list `((a b) c (a b))` matches the pattern `(?x c ?x)` with the pattern variable `?x` bound to `(a b)`. The same data list matches the pattern `(?x ?y ?z)` with `?x` and `?z` both bound to `(a b)` and `?y` bound to `c`. It also matches the pattern `((?x ?y) c (?x ?y))` with `?x` bound to `a` and `?y` bound to `b`. However, it does not match the pattern `(?x a ?y)`, since that pattern specifies a list whose second element is the symbol `a`.

The pattern matcher used by the query system takes as inputs a pattern, a datum, and a *frame* that specifies bindings for various pattern variables. It checks whether the datum matches the pattern in a way that is consistent with the bindings already in the frame. If so, it returns the given frame augmented by any bindings that may have been determined by the match. Otherwise, it indicates that the match has failed.

For example, using the pattern `(?x ?y ?x)` to match `(a b a)` given an empty frame will return a frame specifying that `?x` is bound to `a` and `?y` is bound to `b`. Trying the match with the same pattern, the same datum, and a frame specifying that `?y` is bound to `a` will fail. Trying the

match with the same pattern, the same datum, and a frame in which `?y` is bound to `b` and `?x` is unbound will return the given frame augmented by a binding of `?x` to `a`.

The pattern matcher is all the mechanism that is needed to process simple queries that don't involve rules. For instance, to process the query

```
(job ?x (computer programmer))
```

we scan through all assertions in the data base and select those that match the pattern with respect to an initially empty frame. For each match we find, we use the frame returned by the match to instantiate the pattern with a value for `?x`.

Streams of frames

The testing of patterns against frames is organized through the use of streams. Given a single frame, the matching process runs through the data-base entries one by one. For each data-base entry, the matcher generates either a special symbol indicating that the match has failed or an extension to the frame. The results for all the data-base entries are collected into a stream, which is passed through a filter to weed out the failures. The result is a stream of all the frames that extend the given frame via a match to some assertion in the data base.[67]

[67] Because matching is generally very expensive, we would like to avoid applying the full matcher to every element of the data base. This is usually arranged by breaking up the process into a fast, coarse match and the final match. The coarse match filters the data base to produce a small set of candidates for the final match. With care, we can arrange our data base so that some of the work of coarse matching can be done when the data base is constructed rather then when we want to select the candidates. This is called *indexing* the data base. There is a vast technology built around data-base-indexing schemes. Our implementation, described in Section 4.4.4, contains a simple-minded form of such an optimization.

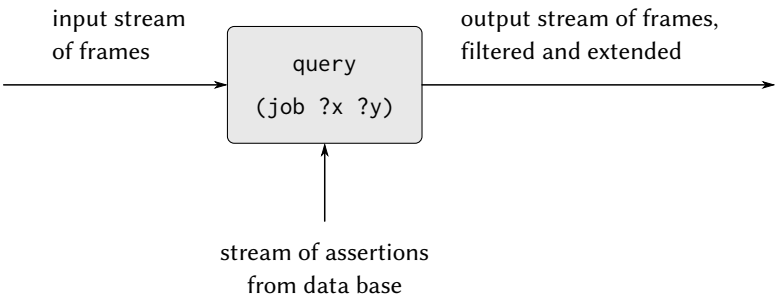

Figure 4.4: A query processes a stream of frames.

In our system, a query takes an input stream of frames and performs the above matching operation for every frame in the stream, as indicated in Figure 4.4. That is, for each frame in the input stream, the query generates a new stream consisting of all extensions to that frame by matches to assertions in the data base. All these streams are then combined to form one huge stream, which contains all possible extensions of every frame in the input stream. This stream is the output of the query.

To answer a simple query, we use the query with an input stream consisting of a single empty frame. The resulting output stream contains all extensions to the empty frame (that is, all answers to our query). This stream of frames is then used to generate a stream of copies of the original query pattern with the variables instantiated by the values in each frame, and this is the stream that is finally printed.

Compound queries

The real elegance of the stream-of-frames implementation is evident when we deal with compound queries. The processing of compound

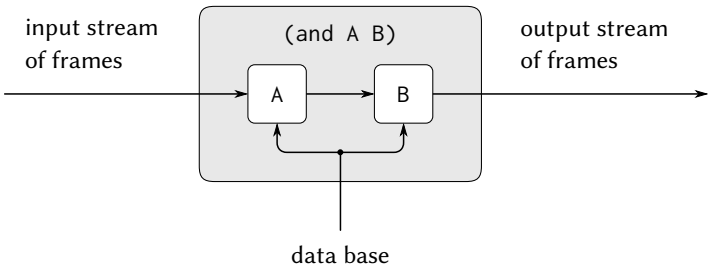

Figure 4.5: The and combination of two queries is produced by operating on the stream of frames in series.

queries makes use of the ability of our matcher to demand that a match be consistent with a specified frame. For example, to handle the and of two queries, such as

```
(and (can-do-job ?x (computer programmer trainee))
     (job ?person ?x))
```

(informally, "Find all people who can do the job of a computer programmer trainee"), we first find all entries that match the pattern

```
(can-do-job ?x (computer programmer trainee))
```

This produces a stream of frames, each of which contains a binding for ?x. Then for each frame in the stream we find all entries that match

```
(job ?person ?x)
```

in a way that is consistent with the given binding for ?x. Each such match will produce a frame containing bindings for ?x and ?person. The and of two queries can be viewed as a series combination of the two component queries, as shown in Figure 4.5. The frames that pass

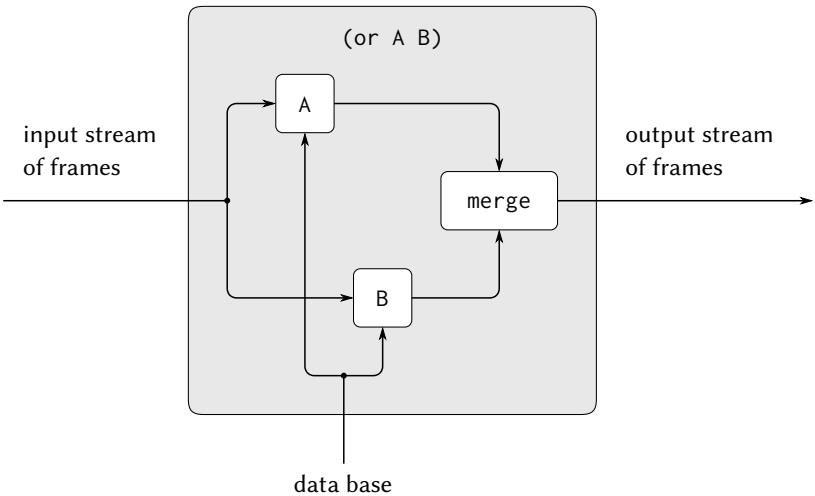

Figure 4.6: The `or` combination of two queries is produced by operating on the stream of frames in parallel and merging the results.

through the first query filter are filtered and further extended by the second query.

Figure 4.6 shows the analogous method for computing the `or` of two queries as a parallel combination of the two component queries. The input stream of frames is extended separately by each query. The two resulting streams are then merged to produce the final output stream.

Even from this high-level description, it is apparent that the processing of compound queries can be slow. For example, since a query may produce more than one output frame for each input frame, and each query in an `and` gets its input frames from the previous query, an `and` query could, in the worst case, have to perform a number of matches

that is exponential in the number of queries (see Exercise 4.76).[68] Though systems for handling only simple queries are quite practical, dealing with complex queries is extremely difficult.[69]

From the stream-of-frames viewpoint, the `not` of some query acts as a filter that removes all frames for which the query can be satisfied. For instance, given the pattern

```
(not (job ?x (computer programmer)))
```

we attempt, for each frame in the input stream, to produce extension frames that satisfy `(job ?x (computer programmer))`. We remove from the input stream all frames for which such extensions exist. The result is a stream consisting of only those frames in which the binding for `?x` does not satisfy `(job ?x (computer programmer))`. For example, in processing the query

```
(and (supervisor ?x ?y)
     (not (job ?x (computer programmer))))
```

the first clause will generate frames with bindings for `?x` and `?y`. The `not` clause will then filter these by removing all frames in which the binding for `?x` satisfies the restriction that `?x` is a computer programmer.[70]

The `lisp-value` special form is implemented as a similar filter on frame streams. We use each frame in the stream to instantiate any variables in the pattern, then apply the Lisp predicate. We remove from the input stream all frames for which the predicate fails.

[68] But this kind of exponential explosion is not common in `and` queries because the added conditions tend to reduce rather than expand the number of frames produced.

[69] There is a large literature on data-base-management systems that is concerned with how to handle complex queries efficiently.

[70] There is a subtle difference between this filter implementation of `not` and the usual meaning of `not` in mathematical logic. See Section 4.4.3.

Unification

In order to handle rules in the query language, we must be able to find the rules whose conclusions match a given query pattern. Rule conclusions are like assertions except that they can contain variables, so we will need a generalization of pattern matching—called *unification*—in which both the "pattern" and the "datum" may contain variables.

A unifier takes two patterns, each containing constants and variables, and determines whether it is possible to assign values to the variables that will make the two patterns equal. If so, it returns a frame containing these bindings. For example, unifying `(?x a ?y)` and `(?y ?z a)` will specify a frame in which `?x`, `?y`, and `?z` must all be bound to `a`. On the other hand, unifying `(?x ?y a)` and `(?x b ?y)` will fail, because there is no value for `?y` that can make the two patterns equal. (For the second elements of the patterns to be equal, `?y` would have to be `b`; however, for the third elements to be equal, `?y` would have to be `a`.) The unifier used in the query system, like the pattern matcher, takes a frame as input and performs unifications that are consistent with this frame.

The unification algorithm is the most technically difficult part of the query system. With complex patterns, performing unification may seem to require deduction. To unify `(?x ?x)` and `((a ?y c) (a b ?z))`, for example, the algorithm must infer that `?x` should be `(a b c)`, `?y` should be `b`, and `?z` should be `c`. We may think of this process as solving a set of equations among the pattern components. In general, these are simultaneous equations, which may require substantial manipulation to solve.[71] For example, unifying `(?x ?x)` and `((a ?y c) (a b ?z))` may be thought of as specifying the simultaneous equations

[71] In one-sided pattern matching, all the equations that contain pattern variables are explicit and already solved for the unknown (the pattern variable).

```
?x  =  (a ?y c)
?x  =  (a b ?z)
```

These equations imply that

```
(a ?y c)  =  (a b ?z)
```

which in turn implies that

```
 a  =  a,
?y  =  b,
 c  =  ?z,
```

and hence that

```
?x  =  (a b c)
```

In a successful pattern match, all pattern variables become bound, and the values to which they are bound contain only constants. This is also true of all the examples of unification we have seen so far. In general, however, a successful unification may not completely determine the variable values; some variables may remain unbound and others may be bound to values that contain variables.

Consider the unification of `(?x a)` and `((b ?y) ?z)`. We can deduce that `?x = (b ?y)` and `a = ?z`, but we cannot further solve for `?x` or `?y`. The unification doesn't fail, since it is certainly possible to make the two patterns equal by assigning values to `?x` and `?y`. Since this match in no way restricts the values `?y` can take on, no binding for `?y` is put into the result frame. The match does, however, restrict the value of `?x`. Whatever value `?y` has, `?x` must be `(b ?y)`. A binding of `?x` to the pattern `(b ?y)` is thus put into the frame. If a value for `?y` is later determined and added to the frame (by a pattern match or unification that is required to be consistent with this frame), the previously bound `?x` will refer to this value.[72]

[72] Another way to think of unification is that it generates the most general pattern

Applying rules

Unification is the key to the component of the query system that makes inferences from rules. To see how this is accomplished, consider processing a query that involves applying a rule, such as

```
(lives-near ?x (Hacker Alyssa P))
```

To process this query, we first use the ordinary pattern-match procedure described above to see if there are any assertions in the data base that match this pattern. (There will not be any in this case, since our data base includes no direct assertions about who lives near whom.) The next step is to attempt to unify the query pattern with the conclusion of each rule. We find that the pattern unifies with the conclusion of the rule

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
           (address ?person-2 (?town . ?rest-2))
           (not (same ?person-1 ?person-2))))
```

resulting in a frame specifying that `?person-2` is bound to `(Hacker Alyssa P)` and that `?x` should be bound to (have the same value as) `?person-1`. Now, relative to this frame, we evaluate the compound query given by the body of the rule. Successful matches will extend this frame by providing a binding for `?person-1`, and consequently a value for `?x`, which we can use to instantiate the original query pattern.

In general, the query evaluator uses the following method to apply a rule when trying to establish a query pattern in a frame that specifies bindings for some of the pattern variables:

---

that is a specialization of the two input patterns. That is, the unification of `(?x a)` and `((b ?y) ?z)` is `((b ?y) a)`, and the unification of `(?x a ?y)` and `(?y ?z a)`, discussed above, is `(a a a)`. For our implementation, it is more convenient to think of the result of unification as a frame rather than a pattern.

- Unify the query with the conclusion of the rule to form, if successful, an extension of the original frame.
- Relative to the extended frame, evaluate the query formed by the body of the rule.

Notice how similar this is to the method for applying a procedure in the `eval/apply` evaluator for Lisp:

- Bind the procedure's parameters to its arguments to form a frame that extends the original procedure environment.
- Relative to the extended environment, evaluate the expression formed by the body of the procedure.

The similarity between the two evaluators should come as no surprise. Just as procedure definitions are the means of abstraction in Lisp, rule definitions are the means of abstraction in the query language. In each case, we unwind the abstraction by creating appropriate bindings and evaluating the rule or procedure body relative to these.

Simple queries

We saw earlier in this section how to evaluate simple queries in the absence of rules. Now that we have seen how to apply rules, we can describe how to evaluate simple queries by using both rules and assertions.

Given the query pattern and a stream of frames, we produce, for each frame in the input stream, two streams:

- a stream of extended frames obtained by matching the pattern against all assertions in the data base (using the pattern matcher), and

- a stream of extended frames obtained by applying all possible rules (using the unifier).[73]

Appending these two streams produces a stream that consists of all the ways that the given pattern can be satisfied consistent with the original frame. These streams (one for each frame in the input stream) are now all combined to form one large stream, which therefore consists of all the ways that any of the frames in the original input stream can be extended to produce a match with the given pattern.

The query evaluator and the driver loop

Despite the complexity of the underlying matching operations, the system is organized much like an evaluator for any language. The procedure that coordinates the matching operations is called `qeval`, and it plays a role analogous to that of the `eval` procedure for Lisp. `qeval` takes as inputs a query and a stream of frames. Its output is a stream of frames, corresponding to successful matches to the query pattern, that extend some frame in the input stream, as indicated in Figure 4.4. Like `eval`, `qeval` classifies the different types of expressions (queries) and dispatches to an appropriate procedure for each. There is a procedure for each special form (`and`, `or`, `not`, and `lisp-value`) and one for simple queries.

The driver loop, which is analogous to the `driver-loop` procedure for the other evaluators in this chapter, reads queries from the terminal. For each query, it calls `qeval` with the query and a stream that consists

[73] Since unification is a generalization of matching, we could simplify the system by using the unifier to produce both streams. Treating the easy case with the simple matcher, however, illustrates how matching (as opposed to full-blown unification) can be useful in its own right.

of a single empty frame. This will produce the stream of all possible matches (all possible extensions to the empty frame). For each frame in the resulting stream, it instantiates the original query using the values of the variables found in the frame. This stream of instantiated queries is then printed.[74]

The driver also checks for the special command `assert!`, which signals that the input is not a query but rather an assertion or rule to be added to the data base. For instance,

```
(assert! (job (Bitdiddle Ben)
              (computer wizard)))
(assert! (rule (wheel ?person)
               (and (supervisor ?middle-manager ?person)
                    (supervisor ?x ?middle-manager))))
```

4.4.3 Is Logic Programming Mathematical Logic?

The means of combination used in the query language may at first seem identical to the operations `and`, `or`, and `not` of mathematical logic, and the application of query-language rules is in fact accomplished through a legitimate method of inference.[75] This identification of the query language with mathematical logic is not really valid, though, because the

[74] The reason we use streams (rather than lists) of frames is that the recursive application of rules can generate infinite numbers of values that satisfy a query. The delayed evaluation embodied in streams is crucial here: The system will print responses one by one as they are generated, regardless of whether there are a finite or infinite number of responses.

[75] That a particular method of inference is legitimate is not a trivial assertion. One must prove that if one starts with true premises, only true conclusions can be derived. The method of inference represented by rule applications is *modus ponens*, the familiar method of inference that says that if A is true and A *implies* B is true, then we may conclude that B is true.

query language provides a *control structure* that interprets the logical statements procedurally. We can often take advantage of this control structure. For example, to find all of the supervisors of programmers we could formulate a query in either of two logically equivalent forms:

```
(and (job ?x (computer programmer)) (supervisor ?x ?y))
```

or

```
(and (supervisor ?x ?y) (job ?x (computer programmer)))
```

If a company has many more supervisors than programmers (the usual case), it is better to use the first form rather than the second because the data base must be scanned for each intermediate result (frame) produced by the first clause of the and.

The aim of logic programming is to provide the programmer with techniques for decomposing a computational problem into two separate problems: "what" is to be computed, and "how" this should be computed. This is accomplished by selecting a subset of the statements of mathematical logic that is powerful enough to be able to describe anything one might want to compute, yet weak enough to have a controllable procedural interpretation. The intention here is that, on the one hand, a program specified in a logic programming language should be an effective program that can be carried out by a computer. Control ("how" to compute) is effected by using the order of evaluation of the language. We should be able to arrange the order of clauses and the order of subgoals within each clause so that the computation is done in an order deemed to be effective and efficient. At the same time, we should be able to view the result of the computation ("what" to compute) as a simple consequence of the laws of logic.

Our query language can be regarded as just such a procedurally interpretable subset of mathematical logic. An assertion represents a sim-

ple fact (an atomic proposition). A rule represents the implication that the rule conclusion holds for those cases where the rule body holds. A rule has a natural procedural interpretation: To establish the conclusion of the rule, establish the body of the rule. Rules, therefore, specify computations. However, because rules can also be regarded as statements of mathematical logic, we can justify any "inference" accomplished by a logic program by asserting that the same result could be obtained by working entirely within mathematical logic.[76]

Infinite loops

A consequence of the procedural interpretation of logic programs is that it is possible to construct hopelessly inefficient programs for solving certain problems. An extreme case of inefficiency occurs when the system falls into infinite loops in making deductions. As a simple example, suppose we are setting up a data base of famous marriages, including

```
(assert! (married Minnie Mickey))
```

If we now ask

```
(married Mickey ?who)
```

[76] We must qualify this statement by agreeing that, in speaking of the "inference" accomplished by a logic program, we assume that the computation terminates. Unfortunately, even this qualified statement is false for our implementation of the query language (and also false for programs in Prolog and most other current logic programming languages) because of our use of `not` and `lisp-value`. As we will describe below, the `not` implemented in the query language is not always consistent with the `not` of mathematical logic, and `lisp-value` introduces additional complications. We could implement a language consistent with mathematical logic by simply removing `not` and `lisp-value` from the language and agreeing to write programs using only simple queries, `and`, and `or`. However, this would greatly restrict the expressive power of the language. One of the major concerns of research in logic programming is to find ways to achieve more consistency with mathematical logic without unduly sacrificing expressive power.

we will get no response, because the system doesn't know that if A is married to B, then B is married to A. So we assert the rule

```
(assert! (rule (married ?x ?y) (married ?y ?x)))
```

and again query

```
(married Mickey ?who)
```

Unfortunately, this will drive the system into an infinite loop, as follows:

- The system finds that the `married` rule is applicable; that is, the rule conclusion `(married ?x ?y)` successfully unifies with the query pattern `(married Mickey ?who)` to produce a frame in which `?x` is bound to `Mickey` and `?y` is bound to `?who`. So the interpreter proceeds to evaluate the rule body `(married ?y ?x)` in this frame—in effect, to process the query `(married ?who Mickey)`.
- One answer appears directly as an assertion in the data base: `(married Minnie Mickey)`.
- The `married` rule is also applicable, so the interpreter again evaluates the rule body, which this time is equivalent to `(married Mickey ?who)`.

The system is now in an infinite loop. Indeed, whether the system will find the simple answer `(married Minnie Mickey)` before it goes into the loop depends on implementation details concerning the order in which the system checks the items in the data base. This is a very simple example of the kinds of loops that can occur. Collections of interrelated rules can lead to loops that are much harder to anticipate, and the appearance of a loop can depend on the order of clauses in an `and` (see

Exercise 4.64) or on low-level details concerning the order in which the system processes queries.[77]

Problems with `not`

Another quirk in the query system concerns `not`. Given the data base of Section 4.4.1, consider the following two queries:

```
(and (supervisor ?x ?y)
     (not (job ?x (computer programmer))))
(and (not (job ?x (computer programmer)))
     (supervisor ?x ?y))
```

These two queries do not produce the same result. The first query begins by finding all entries in the data base that match `(supervisor ?x ?y)`, and then filters the resulting frames by removing the ones in which the value of `?x` satisfies `(job ?x (computer programmer))`. The second query begins by filtering the incoming frames to remove those that can satisfy `(job ?x (computer programmer))`. Since the only incoming frame is empty, it checks the data base to see if there are any patterns that satisfy `(job ?x (computer programmer))`. Since there generally

[77] This is not a problem of the logic but one of the procedural interpretation of the logic provided by our interpreter. We could write an interpreter that would not fall into a loop here. For example, we could enumerate all the proofs derivable from our assertions and our rules in a breadth-first rather than a depth-first order. However, such a system makes it more difficult to take advantage of the order of deductions in our programs. One attempt to build sophisticated control into such a program is described in deKleer et al. 1977. Another technique, which does not lead to such serious control problems, is to put in special knowledge, such as detectors for particular kinds of loops (Exercise 4.67). However, there can be no general scheme for reliably preventing a system from going down infinite paths in performing deductions. Imagine a diabolical rule of the form "To show $P(x)$ is true, show that $P(f(x))$ is true," for some suitably chosen function f.

are entries of this form, the `not` clause filters out the empty frame and returns an empty stream of frames. Consequently, the entire compound query returns an empty stream.

The trouble is that our implementation of `not` really is meant to serve as a filter on values for the variables. If a `not` clause is processed with a frame in which some of the variables remain unbound (as does `?x` in the example above), the system will produce unexpected results. Similar problems occur with the use of `lisp-value`—the Lisp predicate can't work if some of its arguments are unbound. See Exercise 4.77.

There is also a much more serious way in which the `not` of the query language differs from the `not` of mathematical logic. In logic, we interpret the statement "not P" to mean that P is not true. In the query system, however, "not P" means that P is not deducible from the knowledge in the data base. For example, given the personnel data base of Section 4.4.1, the system would happily deduce all sorts of `not` statements, such as that Ben Bitdiddle is not a baseball fan, that it is not raining outside, and that 2 + 2 is not 4.[78] In other words, the `not` of logic programming languages reflects the so-called *closed world assumption* that all relevant information has been included in the data base.[79]

> **Exercise 4.64:** Louis Reasoner mistakenly deletes the `outranked-by` rule (Section 4.4.1) from the data base. When he realizes this, he quickly reinstalls it. Unfortunately, he makes a slight change in the rule, and types it in as

[78] Consider the query `(not (baseball-fan (Bitdiddle Ben)))`. The system finds that `(baseball-fan (Bitdiddle Ben))` is not in the data base, so the empty frame does not satisfy the pattern and is not filtered out of the initial stream of frames. The result of the query is thus the empty frame, which is used to instantiate the input query to produce `(not (baseball-fan (Bitdiddle Ben)))`.

[79] A discussion and justification of this treatment of `not` can be found in the article by Clark (1978).

```
(rule (outranked-by ?staff-person ?boss)
      (or (supervisor ?staff-person ?boss)
          (and (outranked-by ?middle-manager ?boss)
               (supervisor ?staff-person
                           ?middle-manager))))
```

Just after Louis types this information into the system, DeWitt Aull comes by to find out who outranks Ben Bitdiddle. He issues the query

```
(outranked-by (Bitdiddle Ben) ?who)
```

After answering, the system goes into an infinite loop. Explain why.

Exercise 4.65: Cy D. Fect, looking forward to the day when he will rise in the organization, gives a query to find all the wheels (using the `wheel` rule of Section 4.4.1):

```
(wheel ?who)
```

To his surprise, the system responds

```
;;; Query results:
(wheel (Warbucks Oliver))
(wheel (Bitdiddle Ben))
(wheel (Warbucks Oliver))
(wheel (Warbucks Oliver))
(wheel (Warbucks Oliver))
```

Why is Oliver Warbucks listed four times?

Exercise 4.66: Ben has been generalizing the query system to provide statistics about the company. For example,

to find the total salaries of all the computer programmers one will be able to say

```
(sum ?amount (and (job ?x (computer programmer))
                  (salary ?x ?amount)))
```

In general, Ben's new system allows expressions of the form

```
(accumulation-function ⟨variable⟩ ⟨query pattern⟩)
```

where `accumulation-function` can be things like `sum`, `average`, or `maximum`. Ben reasons that it should be a cinch to implement this. He will simply feed the query pattern to `qeval`. This will produce a stream of frames. He will then pass this stream through a mapping function that extracts the value of the designated variable from each frame in the stream and feed the resulting stream of values to the accumulation function. Just as Ben completes the implementation and is about to try it out, Cy walks by, still puzzling over the `wheel` query result in Exercise 4.65. When Cy shows Ben the system's response, Ben groans, "Oh, no, my simple accumulation scheme won't work!"

What has Ben just realized? Outline a method he can use to salvage the situation.

Exercise 4.67: Devise a way to install a loop detector in the query system so as to avoid the kinds of simple loops illustrated in the text and in Exercise 4.64. The general idea is that the system should maintain some sort of history of its current chain of deductions and should not begin processing a query that it is already working on. Describe what kind of information (patterns and frames) is included in

this history, and how the check should be made. (After you study the details of the query-system implementation in Section 4.4.4, you may want to modify the system to include your loop detector.)

Exercise 4.68: Define rules to implement the `reverse` operation of Exercise 2.18, which returns a list containing the same elements as a given list in reverse order. (Hint: Use `append-to-form`.) Can your rules answer both `(reverse (1 2 3) ?x)` and `(reverse ?x (1 2 3))`?

Exercise 4.69: Beginning with the data base and the rules you formulated in Exercise 4.63, devise a rule for adding "greats" to a grandson relationship. This should enable the system to deduce that Irad is the great-grandson of Adam, or that Jabal and Jubal are the great-great-great-great-great-grandsons of Adam. (Hint: Represent the fact about Irad, for example, as `((great grandson) Adam Irad)`. Write rules that determine if a list ends in the word `grandson`. Use this to express a rule that allows one to derive the relationship `((great . ?rel) ?x ?y)`, where `?rel` is a list ending in `grandson`.) Check your rules on queries such as `((great grandson) ?g ?ggs)` and `(?relationship Adam Irad)`.

4.4.4 Implementing the Query System

Section 4.4.2 described how the query system works. Now we fill in the details by presenting a complete implementation of the system.

4.4.4.1 The Driver Loop and Instantiation

The driver loop for the query system repeatedly reads input expressions. If the expression is a rule or assertion to be added to the data base, then the information is added. Otherwise the expression is assumed to be a query. The driver passes this query to the evaluator `qeval` together with an initial frame stream consisting of a single empty frame. The result of the evaluation is a stream of frames generated by satisfying the query with variable values found in the data base. These frames are used to form a new stream consisting of copies of the original query in which the variables are instantiated with values supplied by the stream of frames, and this final stream is printed at the terminal:

```
(define input-prompt  ";;; Query input:")
(define output-prompt ";;; Query results:")

(define (query-driver-loop)
  (prompt-for-input input-prompt)
  (let ((q (query-syntax-process (read))))
    (cond ((assertion-to-be-added? q)
           (add-rule-or-assertion! (add-assertion-body q))
           (newline)
           (display "Assertion added to data base.")
           (query-driver-loop))
          (else
           (newline)
           (display output-prompt)
           (display-stream
            (stream-map
             (lambda (frame)
               (instantiate
                q
                frame
```

```
        (lambda (v f)
          (contract-question-mark v))))
    (qeval q (singleton-stream '()))))
  (query-driver-loop)))))
```

Here, as in the other evaluators in this chapter, we use an abstract syntax for the expressions of the query language. The implementation of the expression syntax, including the predicate `assertion-to-be-added?` and the selector `add-assertion-body`, is given in Section 4.4.4.7. `add-rule-or-assertion!` is defined in Section 4.4.4.5.

Before doing any processing on an input expression, the driver loop transforms it syntactically into a form that makes the processing more efficient. This involves changing the representation of pattern variables. When the query is instantiated, any variables that remain unbound are transformed back to the input representation before being printed. These transformations are performed by the two procedures `query-syntax-process` and `contract-question-mark` (Section 4.4.4.7).

To instantiate an expression, we copy it, replacing any variables in the expression by their values in a given frame. The values are themselves instantiated, since they could contain variables (for example, if `?x` in `exp` is bound to `?y` as the result of unification and `?y` is in turn bound to 5). The action to take if a variable cannot be instantiated is given by a procedural argument to `instantiate`.

```
(define (instantiate exp frame unbound-var-handler)
  (define (copy exp)
    (cond ((var? exp)
           (let ((binding (binding-in-frame exp frame)))
             (if binding
                 (copy (binding-value binding))
                 (unbound-var-handler exp frame))))
          ((pair? exp)
```

```
          (cons (copy (car exp)) (copy (cdr exp))))
         (else exp)))
  (copy exp))
```

The procedures that manipulate bindings are defined in Section 4.4.4.8.

4.4.4.2 The Evaluator

The `qeval` procedure, called by the `query-driver-loop`, is the basic evaluator of the query system. It takes as inputs a query and a stream of frames, and it returns a stream of extended frames. It identifies special forms by a data-directed dispatch using `get` and `put`, just as we did in implementing generic operations in Chapter 2. Any query that is not identified as a special form is assumed to be a simple query, to be processed by `simple-query`.

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
    (if qproc
        (qproc (contents query) frame-stream)
        (simple-query query frame-stream))))
```

`type` and `contents`, defined in Section 4.4.4.7, implement the abstract syntax of the special forms.

Simple queries

The `simple-query` procedure handles simple queries. It takes as arguments a simple query (a pattern) together with a stream of frames, and it returns the stream formed by extending each frame by all data-base matches of the query.

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
```

```
  (lambda (frame)
    (stream-append-delayed
     (find-assertions query-pattern frame)
     (delay (apply-rules query-pattern frame))))
  frame-stream))
```

For each frame in the input stream, we use `find-assertions` (Section 4.4.4.3) to match the pattern against all assertions in the data base, producing a stream of extended frames, and we use `apply-rules` (Section 4.4.4.4) to apply all possible rules, producing another stream of extended frames. These two streams are combined (using `stream-append-delayed`, Section 4.4.4.6) to make a stream of all the ways that the given pattern can be satisfied consistent with the original frame (see Exercise 4.71). The streams for the individual input frames are combined using `stream-flatmap` (Section 4.4.4.6) to form one large stream of all the ways that any of the frames in the original input stream can be extended to produce a match with the given pattern.

Compound queries

`and` queries are handled as illustrated in Figure 4.5 by the `conjoin` procedure. `conjoin` takes as inputs the conjuncts and the frame stream and returns the stream of extended frames. First, `conjoin` processes the stream of frames to find the stream of all possible frame extensions that satisfy the first query in the conjunction. Then, using this as the new frame stream, it recursively applies `conjoin` to the rest of the queries.

```
(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (conjoin (rest-conjuncts conjuncts)
               (qeval (first-conjunct conjuncts) frame-stream))))
```

The expression

```
(put 'and 'qeval conjoin)
```

sets up `qeval` to dispatch to `conjoin` when an `and` form is encountered.

`or` queries are handled similarly, as shown in Figure 4.6. The output streams for the various disjuncts of the `or` are computed separately and merged using the `interleave-delayed` procedure from Section 4.4.4.6. (See Exercise 4.71 and Exercise 4.72.)

```
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave-delayed
       (qeval (first-disjunct disjuncts) frame-stream)
       (delay (disjoin (rest-disjuncts disjuncts) frame-stream)))))
(put 'or 'qeval disjoin)
```

The predicates and selectors for the syntax of conjuncts and disjuncts are given in Section 4.4.4.7.

Filters

`not` is handled by the method outlined in Section 4.4.2. We attempt to extend each frame in the input stream to satisfy the query being negated, and we include a given frame in the output stream only if it cannot be extended.

```
(define (negate operands frame-stream)
  (stream-flatmap
   (lambda (frame)
     (if (stream-null?
          (qeval (negated-query operands)
                 (singleton-stream frame)))
         (singleton-stream frame)
```

```
         the-empty-stream))
   frame-stream))
(put 'not 'qeval negate)
```

`lisp-value` is a filter similar to `not`. Each frame in the stream is used to instantiate the variables in the pattern, the indicated predicate is applied, and the frames for which the predicate returns false are filtered out of the input stream. An error results if there are unbound pattern variables.

```
(define (lisp-value call frame-stream)
  (stream-flatmap
   (lambda (frame)
     (if (execute
          (instantiate
           call
           frame
           (lambda (v f)
             (error "Unknown pat var: LISP-VALUE" v))))
         (singleton-stream frame)
         the-empty-stream))
   frame-stream))
(put 'lisp-value 'qeval lisp-value)
```

`execute`, which applies the predicate to the arguments, must `eval` the predicate expression to get the procedure to apply. However, it must not evaluate the arguments, since they are already the actual arguments, not expressions whose evaluation (in Lisp) will produce the arguments. Note that `execute` is implemented using `eval` and `apply` from the underlying Lisp system.

```
(define (execute exp)
  (apply (eval (predicate exp) user-initial-environment)
         (args exp)))
```

The `always-true` special form provides for a query that is always satisfied. It ignores its contents (normally empty) and simply passes through all the frames in the input stream. `always-true` is used by the `rule-body` selector (Section 4.4.4.7) to provide bodies for rules that were defined without bodies (that is, rules whose conclusions are always satisfied).

```
(define (always-true ignore frame-stream) frame-stream)
(put 'always-true 'qeval always-true)
```

The selectors that define the syntax of `not` and `lisp-value` are given in Section 4.4.4.7.

4.4.4.3 Finding Assertions by Pattern Matching

`find-assertions`, called by `simple-query` (Section 4.4.4.2), takes as input a pattern and a frame. It returns a stream of frames, each extending the given one by a data-base match of the given pattern. It uses `fetch-assertions` (Section 4.4.4.5) to get a stream of all the assertions in the data base that should be checked for a match against the pattern and the frame. The reason for `fetch-assertions` here is that we can often apply simple tests that will eliminate many of the entries in the data base from the pool of candidates for a successful match. The system would still work if we eliminated `fetch-assertions` and simply checked a stream of all assertions in the data base, but the computation would be less efficient because we would need to make many more calls to the matcher.

```
(define (find-assertions pattern frame)
  (stream-flatmap
   (lambda (datum) (check-an-assertion datum pattern frame))
   (fetch-assertions pattern frame)))
```

`check-an-assertion` takes as arguments a pattern, a data object (assertion), and a frame and returns either a one-element stream containing the extended frame or `the-empty-stream` if the match fails.

```
(define (check-an-assertion assertion query-pat query-frame)
  (let ((match-result
         (pattern-match query-pat assertion query-frame)))
    (if (eq? match-result 'failed)
        the-empty-stream
        (singleton-stream match-result))))
```

The basic pattern matcher returns either the symbol `failed` or an extension of the given frame. The basic idea of the matcher is to check the pattern against the data, element by element, accumulating bindings for the pattern variables. If the pattern and the data object are the same, the match succeeds and we return the frame of bindings accumulated so far. Otherwise, if the pattern is a variable we extend the current frame by binding the variable to the data, so long as this is consistent with the bindings already in the frame. If the pattern and the data are both pairs, we (recursively) match the `car` of the pattern against the `car` of the data to produce a frame; in this frame we then match the `cdr` of the pattern against the `cdr` of the data. If none of these cases are applicable, the match fails and we return the symbol `failed`.

```
(define (pattern-match pat dat frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? pat dat) frame)
        ((var? pat) (extend-if-consistent pat dat frame))
        ((and (pair? pat) (pair? dat))
         (pattern-match
          (cdr pat)
          (cdr dat)
          (pattern-match (car pat) (car dat) frame)))
```

```
        (else 'failed)))
```

Here is the procedure that extends a frame by adding a new binding, if this is consistent with the bindings already in the frame:

```
(define (extend-if-consistent var dat frame)
  (let ((binding (binding-in-frame var frame)))
    (if binding
        (pattern-match (binding-value binding) dat frame)
        (extend var dat frame))))
```

If there is no binding for the variable in the frame, we simply add the binding of the variable to the data. Otherwise we match, in the frame, the data against the value of the variable in the frame. If the stored value contains only constants, as it must if it was stored during pattern matching by `extend-if-consistent`, then the match simply tests whether the stored and new values are the same. If so, it returns the unmodified frame; if not, it returns a failure indication. The stored value may, however, contain pattern variables if it was stored during unification (see Section 4.4.4.4). The recursive match of the stored pattern against the new data will add or check bindings for the variables in this pattern. For example, suppose we have a frame in which `?x` is bound to `(f ?y)` and `?y` is unbound, and we wish to augment this frame by a binding of `?x` to `(f b)`. We look up `?x` and find that it is bound to `(f ?y)`. This leads us to match `(f ?y)` against the proposed new value `(f b)` in the same frame. Eventually this match extends the frame by adding a binding of `?y` to `b`. `?X` remains bound to `(f ?y)`. We never modify a stored binding and we never store more than one binding for a given variable.

The procedures used by `extend-if-consistent` to manipulate bindings are defined in Section 4.4.4.8.

Patterns with dotted tails

If a pattern contains a dot followed by a pattern variable, the pattern variable matches the rest of the data list (rather than the next element of the data list), just as one would expect with the dotted-tail notation described in Exercise 2.20. Although the pattern matcher we have just implemented doesn't look for dots, it does behave as we want. This is because the Lisp `read` primitive, which is used by `query-driver-loop` to read the query and represent it as a list structure, treats dots in a special way.

When `read` sees a dot, instead of making the next item be the next element of a list (the `car` of a `cons` whose `cdr` will be the rest of the list) it makes the next item be the `cdr` of the list structure. For example, the list structure produced by `read` for the pattern `(computer ?type)` could be constructed by evaluating the expression `(cons 'computer (cons '?type '()))`, and that for `(computer . ?type)` could be constructed by evaluating the expression `(cons 'computer '?type)`.

Thus, as `pattern-match` recursively compares `cars` and `cdrs` of a data list and a pattern that had a dot, it eventually matches the variable after the dot (which is a `cdr` of the pattern) against a sublist of the data list, binding the variable to that list. For example, matching the pattern `(computer . ?type)` against `(computer programmer trainee)` will match `?type` against the list `(programmer trainee)`.

4.4.4.4 Rules and Unification

`apply-rules` is the rule analog of `find-assertions` (Section 4.4.4.3). It takes as input a pattern and a frame, and it forms a stream of extension frames by applying rules from the data base. `stream-flatmap` maps `apply-a-rule` down the stream of possibly applicable rules (selected

by `fetch-rules`, Section 4.4.4.5) and combines the resulting streams of frames.

```
(define (apply-rules pattern frame)
  (stream-flatmap (lambda (rule)
                    (apply-a-rule rule pattern frame))
                  (fetch-rules pattern frame)))
```

`apply-a-rule` applies rules using the method outlined in Section 4.4.2. It first augments its argument frame by unifying the rule conclusion with the pattern in the given frame. If this succeeds, it evaluates the rule body in this new frame.

Before any of this happens, however, the program renames all the variables in the rule with unique new names. The reason for this is to prevent the variables for different rule applications from becoming confused with each other. For instance, if two rules both use a variable named `?x`, then each one may add a binding for `?x` to the frame when it is applied. These two `?x`'s have nothing to do with each other, and we should not be fooled into thinking that the two bindings must be consistent. Rather than rename variables, we could devise a more clever environment structure; however, the renaming approach we have chosen here is the most straightforward, even if not the most efficient. (See Exercise 4.79.) Here is the `apply-a-rule` procedure:

```
(define (apply-a-rule rule query-pattern query-frame)
  (let ((clean-rule (rename-variables-in rule)))
    (let ((unify-result (unify-match query-pattern
                                     (conclusion clean-rule)
                                     query-frame)))
      (if (eq? unify-result 'failed)
          the-empty-stream
          (qeval (rule-body clean-rule)
                 (singleton-stream unify-result))))))
```

The selectors `rule-body` and `conclusion` that extract parts of a rule are defined in Section 4.4.4.7.

We generate unique variable names by associating a unique identifier (such as a number) with each rule application and combining this identifier with the original variable names. For example, if the rule-application identifier is 7, we might change each `?x` in the rule to `?x-7` and each `?y` in the rule to `?y-7`. (`make-new-variable` and `new-rule-application-id` are included with the syntax procedures in Section 4.4.4.7.)

```
(define (rename-variables-in rule)
  (let ((rule-application-id (new-rule-application-id)))
    (define (tree-walk exp)
      (cond ((var? exp)
             (make-new-variable exp rule-application-id))
            ((pair? exp)
             (cons (tree-walk (car exp))
                   (tree-walk (cdr exp))))
            (else exp)))
    (tree-walk rule)))
```

The unification algorithm is implemented as a procedure that takes as inputs two patterns and a frame and returns either the extended frame or the symbol `failed`. The unifier is like the pattern matcher except that it is symmetrical—variables are allowed on both sides of the match. `unify-match` is basically the same as `pattern-match`, except that there is extra code (marked "`***`" below) to handle the case where the object on the right side of the match is a variable.

```
(define (unify-match p1 p2 frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? p1 p2) frame)
        ((var? p1) (extend-if-possible p1 p2 frame))
```

```
((var? p2) (extend-if-possible p2 p1 frame))  ;***
((and (pair? p1) (pair? p2))
 (unify-match (cdr p1)
              (cdr p2)
              (unify-match (car p1)
                           (car p2)
                           frame)))
(else 'failed)))
```

In unification, as in one-sided pattern matching, we want to accept a proposed extension of the frame only if it is consistent with existing bindings. The procedure `extend-if-possible` used in unification is the same as the `extend-if-consistent` used in pattern matching except for two special checks, marked "`***`" in the program below. In the first case, if the variable we are trying to match is not bound, but the value we are trying to match it with is itself a (different) variable, it is necessary to check to see if the value is bound, and if so, to match its value. If both parties to the match are unbound, we may bind either to the other.

The second check deals with attempts to bind a variable to a pattern that includes that variable. Such a situation can occur whenever a variable is repeated in both patterns. Consider, for example, unifying the two patterns `(?x ?x)` and `(?y` ⟨*expression involving ?y*⟩`)` in a frame where both `?x` and `?y` are unbound. First `?x` is matched against `?y`, making a binding of `?x` to `?y`. Next, the same `?x` is matched against the given expression involving `?y`. Since `?x` is already bound to `?y`, this results in matching `?y` against the expression. If we think of the unifier as finding a set of values for the pattern variables that make the patterns the same, then these patterns imply instructions to find a `?y` such that `?y` is equal to the expression involving `?y`. There is no general method for solving such equations, so we reject such bindings; these cases are

recognized by the predicate `depends-on?`.[80] On the other hand, we do not want to reject attempts to bind a variable to itself. For example, consider unifying `(?x ?x)` and `(?y ?y)`. The second attempt to bind `?x` to `?y` matches `?y` (the stored value of `?x`) against `?y` (the new value of `?x`). This is taken care of by the `equal?` clause of `unify-match`.

```
(define (extend-if-possible var val frame)
  (let ((binding (binding-in-frame var frame)))
    (cond (binding
```

[80] In general, unifying `?y` with an expression involving `?y` would require our being able to find a fixed point of the equation `?y` = ⟨*expression involving ?y*⟩. It is sometimes possible to syntactically form an expression that appears to be the solution. For example, `?y` = `(f ?y)` seems to have the fixed point `(f (f (f ... )))`, which we can produce by beginning with the expression `(f ?y)` and repeatedly substituting `(f ?y)` for `?y`. Unfortunately, not every such equation has a meaningful fixed point. The issues that arise here are similar to the issues of manipulating infinite series in mathematics. For example, we know that 2 is the solution to the equation $y = 1 + y/2$. Beginning with the expression $1 + y/2$ and repeatedly substituting $1 + y/2$ for y gives

$$2 = y = 1 + \frac{y}{2} = 1 + \frac{1}{2}\left(1 + \frac{y}{2}\right) = 1 + \frac{1}{2} + \frac{y}{4} = \ldots,$$

which leads to

$$2 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots.$$

However, if we try the same manipulation beginning with the observation that -1 is the solution to the equation $y = 1 + 2y$, we obtain

$$-1 = y = 1 + 2y = 1 + 2(1 + 2y) = 1 + 2 + 4y = \ldots,$$

which leads to

$$-1 = 1 + 2 + 4 + 8 + \ldots.$$

Although the formal manipulations used in deriving these two equations are identical, the first result is a valid assertion about infinite series but the second is not. Similarly, for our unification results, reasoning with an arbitrary syntactically constructed expression may lead to errors.

```
         (unify-match (binding-value binding) val frame))
        ((var? val)                          ;***
         (let ((binding (binding-in-frame val frame)))
           (if binding
               (unify-match
                var (binding-value binding) frame)
               (extend var val frame))))
        ((depends-on? val var frame)         ;***
         'failed)
        (else (extend var val frame)))))
```

depends-on? is a predicate that tests whether an expression proposed to be the value of a pattern variable depends on the variable. This must be done relative to the current frame because the expression may contain occurrences of a variable that already has a value that depends on our test variable. The structure of depends-on? is a simple recursive tree walk in which we substitute for the values of variables whenever necessary.

```
(define (depends-on? exp var frame)
  (define (tree-walk e)
    (cond ((var? e)
           (if (equal? var e)
               true
               (let ((b (binding-in-frame e frame)))
                 (if b
                     (tree-walk (binding-value b))
                     false))))
          ((pair? e)
           (or (tree-walk (car e))
               (tree-walk (cdr e))))
          (else false)))
  (tree-walk exp))
```

4.4.4.5 Maintaining the Data Base

One important problem in designing logic programming languages is that of arranging things so that as few irrelevant data-base entries as possible will be examined in checking a given pattern. In our system, in addition to storing all assertions in one big stream, we store all assertions whose `car`s are constant symbols in separate streams, in a table indexed by the symbol. To fetch an assertion that may match a pattern, we first check to see if the `car` of the pattern is a constant symbol. If so, we return (to be tested using the matcher) all the stored assertions that have the same `car`. If the pattern's `car` is not a constant symbol, we return all the stored assertions. Cleverer methods could also take advantage of information in the frame, or try also to optimize the case where the `car` of the pattern is not a constant symbol. We avoid building our criteria for indexing (using the `car`, handling only the case of constant symbols) into the program; instead we call on predicates and selectors that embody our criteria.

```
(define THE-ASSERTIONS the-empty-stream)
(define (fetch-assertions pattern frame)
  (if (use-index? pattern)
      (get-indexed-assertions pattern)
      (get-all-assertions)))
(define (get-all-assertions) THE-ASSERTIONS)
(define (get-indexed-assertions pattern)
  (get-stream (index-key-of pattern) 'assertion-stream))
```

`get-stream` looks up a stream in the table and returns an empty stream if nothing is stored there.

```
(define (get-stream key1 key2)
  (let ((s (get key1 key2)))
    (if s s the-empty-stream)))
```

Rules are stored similarly, using the `car` of the rule conclusion. Rule conclusions are arbitrary patterns, however, so they differ from assertions in that they can contain variables. A pattern whose `car` is a constant symbol can match rules whose conclusions start with a variable as well as rules whose conclusions have the same `car`. Thus, when fetching rules that might match a pattern whose `car` is a constant symbol we fetch all rules whose conclusions start with a variable as well as those whose conclusions have the same `car` as the pattern. For this purpose we store all rules whose conclusions start with a variable in a separate stream in our table, indexed by the symbol `?`.

```
(define THE-RULES the-empty-stream)
(define (fetch-rules pattern frame)
  (if (use-index? pattern)
      (get-indexed-rules pattern)
      (get-all-rules)))
(define (get-all-rules) THE-RULES)
(define (get-indexed-rules pattern)
  (stream-append
   (get-stream (index-key-of pattern) 'rule-stream)
   (get-stream '? 'rule-stream)))
```

`add-rule-or-assertion!` is used by `query-driver-loop` to add assertions and rules to the data base. Each item is stored in the index, if appropriate, and in a stream of all assertions or rules in the data base.

```
(define (add-rule-or-assertion! assertion)
  (if (rule? assertion)
      (add-rule! assertion)
      (add-assertion! assertion)))
(define (add-assertion! assertion)
  (store-assertion-in-index assertion)
  (let ((old-assertions THE-ASSERTIONS))
```

```
    (set! THE-ASSERTIONS
          (cons-stream assertion old-assertions))
    'ok))
(define (add-rule! rule)
  (store-rule-in-index rule)
  (let ((old-rules THE-RULES))
    (set! THE-RULES (cons-stream rule old-rules))
    'ok))
```

To actually store an assertion or a rule, we check to see if it can be indexed. If so, we store it in the appropriate stream.

```
(define (store-assertion-in-index assertion)
  (if (indexable? assertion)
      (let ((key (index-key-of assertion)))
        (let ((current-assertion-stream
               (get-stream key 'assertion-stream)))
          (put key
               'assertion-stream
               (cons-stream
                assertion
                current-assertion-stream))))))
(define (store-rule-in-index rule)
  (let ((pattern (conclusion rule)))
    (if (indexable? pattern)
        (let ((key (index-key-of pattern)))
          (let ((current-rule-stream
                 (get-stream key 'rule-stream)))
            (put key
                 'rule-stream
                 (cons-stream rule
                              current-rule-stream)))))))
```

The following procedures define how the data-base index is used. A pattern (an assertion or a rule conclusion) will be stored in the table if it

starts with a variable or a constant symbol.

```
(define (indexable? pat)
  (or (constant-symbol? (car pat))
      (var? (car pat))))
```

The key under which a pattern is stored in the table is either ? (if it starts with a variable) or the constant symbol with which it starts.

```
(define (index-key-of pat)
  (let ((key (car pat)))
    (if (var? key) '? key)))
```

The index will be used to retrieve items that might match a pattern if the pattern starts with a constant symbol.

```
(define (use-index? pat) (constant-symbol? (car pat)))
```

> **Exercise 4.70:** What is the purpose of the `let` bindings in the procedures `add-assertion!` and `add-rule!` ? What would be wrong with the following implementation of `add-assertion!` ? Hint: Recall the definition of the infinite stream of ones in Section 3.5.2: `(define ones (cons-stream 1 ones))`.
>
> ```
> (define (add-assertion! assertion)
>   (store-assertion-in-index assertion)
>   (set! THE-ASSERTIONS
>         (cons-stream assertion THE-ASSERTIONS))
>   'ok)
> ```

4.4.4.6 Stream Operations

The query system uses a few stream operations that were not presented in Chapter 3.

`stream-append-delayed` and `interleave-delayed` are just like `stream-append` and `interleave` (Section 3.5.3), except that they take a delayed argument (like the `integral` procedure in Section 3.5.4). This postpones looping in some cases (see Exercise 4.71).

```
(define (stream-append-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
       (stream-car s1)
       (stream-append-delayed
        (stream-cdr s1)
        delayed-s2))))
(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
       (stream-car s1)
       (interleave-delayed
        (force delayed-s2)
        (delay (stream-cdr s1))))))
```

`stream-flatmap`, which is used throughout the query evaluator to map a procedure over a stream of frames and combine the resulting streams of frames, is the stream analog of the `flatmap` procedure introduced for ordinary lists in Section 2.2.3. Unlike ordinary `flatmap`, however, we accumulate the streams with an interleaving process, rather than simply appending them (see Exercise 4.72 and Exercise 4.73).

```
(define (stream-flatmap proc s)
  (flatten-stream (stream-map proc s)))

(define (flatten-stream stream)
  (if (stream-null? stream)
```

```
      the-empty-stream
      (interleave-delayed
       (stream-car stream)
       (delay (flatten-stream (stream-cdr stream))))))
```

The evaluator also uses the following simple procedure to generate a stream consisting of a single element:

```
(define (singleton-stream x)
  (cons-stream x the-empty-stream))
```

4.4.4.7 Query Syntax Procedures

`type` and `contents`, used by `qeval` (Section 4.4.4.2), specify that a special form is identified by the symbol in its `car`. They are the same as the `type-tag` and `contents` procedures in Section 2.4.2, except for the error message.

```
(define (type exp)
  (if (pair? exp)
      (car exp)
      (error "Unknown expression TYPE" exp)))
(define (contents exp)
  (if (pair? exp)
      (cdr exp)
      (error "Unknown expression CONTENTS" exp)))
```

The following procedures, used by `query-driver-loop` (in Section 4.4.4.1), specify that rules and assertions are added to the data base by expressions of the form `(assert!` ⟨*rule-or-assertion*⟩`)`:

```
(define (assertion-to-be-added? exp)
  (eq? (type exp) 'assert!))
(define (add-assertion-body exp) (car (contents exp)))
```

Here are the syntax definitions for the `and`, `or`, `not`, and `lisp-value` special forms (Section 4.4.4.2):

```
(define (empty-conjunction? exps) (null? exps))
(define (first-conjunct exps) (car exps))
(define (rest-conjuncts exps) (cdr exps))
(define (empty-disjunction? exps) (null? exps))
(define (first-disjunct exps) (car exps))
(define (rest-disjuncts exps) (cdr exps))
(define (negated-query exps) (car exps))
(define (predicate exps) (car exps))
(define (args exps) (cdr exps))
```

The following three procedures define the syntax of rules:

```
(define (rule? statement)
  (tagged-list? statement 'rule))
(define (conclusion rule) (cadr rule))
(define (rule-body rule)
  (if (null? (cddr rule)) '(always-true) (caddr rule)))
```

`query-driver-loop` (Section 4.4.4.1) calls `query-syntax-process` to transform pattern variables in the expression, which have the form `?symbol`, into the internal format `(? symbol)`. That is to say, a pattern such as `(job ?x ?y)` is actually represented internally by the system as `(job (? x) (? y))`. This increases the efficiency of query processing, since it means that the system can check to see if an expression is a pattern variable by checking whether the `car` of the expression is the symbol `?`, rather than having to extract characters from the symbol. The syntax transformation is accomplished by the following procedure:[81]

[81] Most Lisp systems give the user the ability to modify the ordinary `read` procedure to perform such transformations by defining *reader macro characters*. Quoted expressions are already handled in this way: The reader automatically translates

```
(define (query-syntax-process exp)
  (map-over-symbols expand-question-mark exp))
(define (map-over-symbols proc exp)
  (cond ((pair? exp)
         (cons (map-over-symbols proc (car exp))
               (map-over-symbols proc (cdr exp))))
        ((symbol? exp) (proc exp))
        (else exp)))
(define (expand-question-mark symbol)
  (let ((chars (symbol->string symbol)))
    (if (string=? (substring chars 0 1) "?")
        (list '?
              (string->symbol
               (substring chars 1 (string-length chars))))
        symbol)))
```

Once the variables are transformed in this way, the variables in a pattern are lists starting with ?, and the constant symbols (which need to be recognized for data-base indexing, Section 4.4.4.5) are just the symbols.

```
(define (var? exp) (tagged-list? exp '?))
(define (constant-symbol? exp) (symbol? exp))
```

Unique variables are constructed during rule application (in Section 4.4.4.4) by means of the following procedures. The unique identifier for a rule application is a number, which is incremented each time a rule is applied.

```
(define rule-counter 0)
```

`'expression` into `(quote expression)` before the evaluator sees it. We could arrange for `?expression` to be transformed into `(? expression)` in the same way; however, for the sake of clarity we have included the transformation procedure here explicitly.

`expand-question-mark` and `contract-question-mark` use several procedures with `string` in their names. These are Scheme primitives.

```
(define (new-rule-application-id)
  (set! rule-counter (+ 1 rule-counter))
  rule-counter)
(define (make-new-variable var rule-application-id)
  (cons '? (cons rule-application-id (cdr var))))
```

When `query-driver-loop` instantiates the query to print the answer, it converts any unbound pattern variables back to the right form for printing, using

```
(define (contract-question-mark variable)
  (string->symbol
   (string-append "?"
     (if (number? (cadr variable))
         (string-append (symbol->string (caddr variable))
                        "_"
                        (number->string (cadr variable)))
         (symbol->string (cadr variable))))))
```

4.4.4.8 Frames and Bindings

Frames are represented as lists of bindings, which are variable-value pairs:

```
(define (make-binding variable value)
  (cons variable value))
(define (binding-variable binding) (car binding))
(define (binding-value binding) (cdr binding))
(define (binding-in-frame variable frame)
  (assoc variable frame))
(define (extend variable value frame)
  (cons (make-binding variable value) frame))
```

Exercise 4.71: Louis Reasoner wonders why the `simple-query` and `disjoin` procedures (Section 4.4.4.2) are implemented using explicit `delay` operations, rather than being defined as follows:

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append
      (find-assertions query-pattern frame)
      (apply-rules query-pattern frame)))
   frame-stream))
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave
       (qeval (first-disjunct disjuncts)
              frame-stream)
       (disjoin (rest-disjuncts disjuncts)
                frame-stream))))
```

Can you give examples of queries where these simpler definitions would lead to undesirable behavior?

Exercise 4.72: Why do `disjoin` and `stream-flatmap` interleave the streams rather than simply append them? Give examples that illustrate why interleaving works better. (Hint: Why did we use `interleave` in Section 3.5.3?)

Exercise 4.73: Why does `flatten-stream` use `delay` explicitly? What would be wrong with defining it as follows:

```
(define (flatten-stream stream)
```

```
(if (stream-null? stream)
    the-empty-stream
    (interleave
     (stream-car stream)
     (flatten-stream (stream-cdr stream)))))
```

Exercise 4.74: Alyssa P. Hacker proposes to use a simpler version of `stream-flatmap` in `negate`, `lisp-value`, and `find-assertions`. She observes that the procedure that is mapped over the frame stream in these cases always produces either the empty stream or a singleton stream, so no interleaving is needed when combining these streams.

a. Fill in the missing expressions in Alyssa's program.

```
(define (simple-stream-flatmap proc s)
  (simple-flatten (stream-map proc s)))
(define (simple-flatten stream)
  (stream-map ⟨??⟩
              (stream-filter ⟨??⟩ stream)))
```

b. Does the query system's behavior change if we change it in this way?

Exercise 4.75: Implement for the query language a new special form called `unique`. `unique` should succeed if there is precisely one item in the data base satisfying a specified query. For example,

```
(unique (job ?x (computer wizard)))
```

should print the one-item stream

```
(unique (job (Bitdiddle Ben) (computer wizard)))
```

since Ben is the only computer wizard, and

```
(unique (job ?x (computer programmer)))
```

should print the empty stream, since there is more than one computer programmer. Moreover,

```
(and (job ?x ?j) (unique (job ?anyone ?j)))
```

should list all the jobs that are filled by only one person, and the people who fill them.

There are two parts to implementing `unique`. The first is to write a procedure that handles this special form, and the second is to make `qeval` dispatch to that procedure. The second part is trivial, since `qeval` does its dispatching in a data-directed way. If your procedure is called `uniquely-asserted`, all you need to do is

```
(put 'unique 'qeval uniquely-asserted)
```

and `qeval` will dispatch to this procedure for every query whose `type` (`car`) is the symbol `unique`.

The real problem is to write the procedure `uniquely-asserted`. This should take as input the `contents` (`cdr`) of the `unique` query, together with a stream of frames. For each frame in the stream, it should use `qeval` to find the stream of all extensions to the frame that satisfy the given query. Any stream that does not have exactly one item in it should be eliminated. The remaining streams should be passed back to be accumulated into one big stream that is the result of the `unique` query. This is similar to the implementation of the `not` special form.

Test your implementation by forming a query that lists all people who supervise precisely one person.

Exercise 4.76: Our implementation of `and` as a series combination of queries (Figure 4.5) is elegant, but it is inefficient because in processing the second query of the `and` we must scan the data base for each frame produced by the first query. If the data base has n elements, and a typical query produces a number of output frames proportional to n (say n/k), then scanning the data base for each frame produced by the first query will require n^2/k calls to the pattern matcher. Another approach would be to process the two clauses of the `and` separately, then look for all pairs of output frames that are compatible. If each query produces n/k output frames, then this means that we must perform n^2/k^2 compatibility checks—a factor of k fewer than the number of matches required in our current method.

Devise an implementation of `and` that uses this strategy. You must implement a procedure that takes two frames as inputs, checks whether the bindings in the frames are compatible, and, if so, produces a frame that merges the two sets of bindings. This operation is similar to unification.

Exercise 4.77: In Section 4.4.3 we saw that `not` and `lisp-value` can cause the query language to give "wrong" answers if these filtering operations are applied to frames in which variables are unbound. Devise a way to fix this shortcoming. One idea is to perform the filtering in a "delayed" manner by appending to the frame a "promise" to filter that is fulfilled only when enough variables have been bound

to make the operation possible. We could wait to perform filtering until all other operations have been performed. However, for efficiency's sake, we would like to perform filtering as soon as possible so as to cut down on the number of intermediate frames generated.

Exercise 4.78: Redesign the query language as a nondeterministic program to be implemented using the evaluator of Section 4.3, rather than as a stream process. In this approach, each query will produce a single answer (rather than the stream of all answers) and the user can type `try-again` to see more answers. You should find that much of the mechanism we built in this section is subsumed by nondeterministic search and backtracking. You will probably also find, however, that your new query language has subtle differences in behavior from the one implemented here. Can you find examples that illustrate this difference?

Exercise 4.79: When we implemented the Lisp evaluator in Section 4.1, we saw how to use local environments to avoid name conflicts between the parameters of procedures. For example, in evaluating

```
(define (square x) (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(sum-of-squares 3 4)
```

there is no confusion between the `x` in `square` and the `x` in `sum-of-squares`, because we evaluate the body of each procedure in an environment that is specially constructed

to contain bindings for the local variables. In the query system, we used a different strategy to avoid name conflicts in applying rules. Each time we apply a rule we rename the variables with new names that are guaranteed to be unique. The analogous strategy for the Lisp evaluator would be to do away with local environments and simply rename the variables in the body of a procedure each time we apply the procedure.

Implement for the query language a rule-application method that uses environments rather than renaming. See if you can build on your environment structure to create constructs in the query language for dealing with large systems, such as the rule analog of block-structured procedures. Can you relate any of this to the problem of making deductions in a context (e.g., "If I supposed that *P* were true, then I would be able to deduce *A* and *B*.") as a method of problem solving? (This problem is open-ended. A good answer is probably worth a Ph.D.)