17 Lisp-shell: An Expert System Shell in Lisp

Chapter Objectives

This chapter defines streams (lists) and delayed evaluation with functions
    `delay`
    `force`
Stream processing based on lexical closures
    Freezes evaluation of stream
    Closures preserve variable bindings and scope
`lisp-shell` created as full expert system shell in Lisp
Unification based on stream processing
Askable list organizes user queries
    Full certainty factor system based on Stanford Certainty Factor Algebra
Demonstration of `lisp-shell` with a "plant identification" data base
Exercises on extending function of `lisp-shell`

Chapter Contents

17.1 Streams and Delayed Evaluation
17.2 An Expert System Shell in Lisp

17.1 Streams and Delayed Evaluation

Why delayed evaluation?

As we demonstrated in the implementation of `logic-shell` in Chapter 16, a stream-oriented view can help with the organization of a complex program. However, our implementation of streams as lists did not provide the full benefit of stream processing. In particular, this implementation suffers from inefficiency and an inability to handle potentially infinite data streams.

In the list implementation of streams, all of the elements must be computed before that stream (list) can be passed on to the next function. In `logic-shell` this leads to an exhaustive search of the knowledge base for each intermediate goal in the solution process. In order to produce the first solution to the top-level goal, the program must produce a list of all solutions. Even if we want only the first solution on this list, the program must still search the entire solution space. What we would really prefer is for the program to produce just the first solution by searching only that portion of the space needed to produce that solution and then to delay finding the rest of the goals until they are needed.

A second problem is the inability to process potentially infinite streams of information. Although this problem does not arise in `logic-shell`, it occurs naturally in the stream-based solution to many problems. Assume, for example, that we would like to write a function that returns a stream of the first n odd Fibonacci numbers. A straightforward implementation would use a generator to produce a stream of Fibonacci numbers, a filter to eliminate the

even-valued numbers from the stream, and an accumulator to gather these into a solution list of n elements, as in Figure 17.1. Unfortunately, the stream of Fibonacci numbers is infinite in length and we cannot decide in advance how long a list will be needed to produce the first n odd numbers.

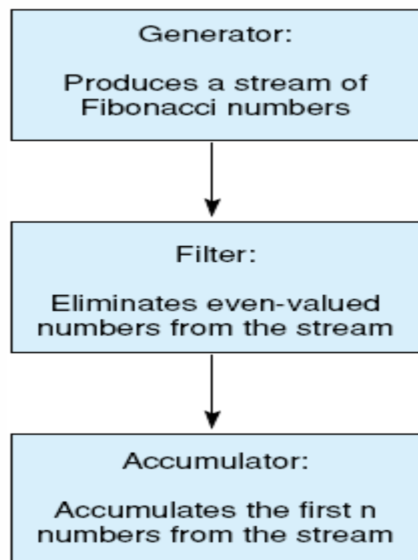

Figure 17.1. A stream implementation that finds the first n odd Fibonacci numbers.

Instead, we would like the generator to produce the stream of Fibonacci numbers one at a time and pass each number through the filter until the accumulator has gathered the n values required. This behavior more closely fits our intuitive notion of evaluating a stream than does the list-based implementation of Chapter 16. We accomplish this by use of *delayed evaluation*.

Delayed Evaluation and Function Closures

Instead of letting the generator run to completion to produce the entire stream of results, we let the function produce the first element of the stream and then freeze or delay its execution until the next element is needed. When the program needs the next element of the stream, it causes the function to resume execution and produce only that element and again delay evaluation of the rest of the stream. Thus, instead of containing the entire list of numbers, the stream consists of just two components, its first element and the frozen computation of the rest of the stream, as shown in Figure 17.2.

A list-based stream containing an indeterminate number of elements:

$(e_1\ e_2\ e_3\ e_4\ .\ .\ .)$

A stream with delayed evaluation of its tail containing two elements but capable of producing any number of elements:

(e_1 . <delayed evaluation of rest of stream>)

Figure 17.2 A list-based and delayed evaluation of streams.

We use *function closures* to create the delayed portion of the stream that was illustrated by Figure 16.1. A closure consists of a function, along with all its variable bindings in the current environment; we may bind a closure to a variable, or pass it as a parameter, and evaluate it using `funcall`. Essentially, a closure "freezes" a function application until a later time. We can create closures using the Lisp form `function`. For example, consider the following Lisp transcript:

```
> (setq v 10)
10
> (let ((v 20)) (setq f_closure (function (lambda (
) v))))
#<COMPILED-LEXICAL-CLOSURE #x28641E>
> (funcall f_closure)
20
>
10
```

The initial `setq` binds `v` to `10` in the global environment. In the `let` block, we create a local binding of `v` to `20` and create a closure of a function that returns this value of `v`. It is interesting to note that this binding of `v` does not disappear when we exit the `let` block, because it is retained in the function closure that is bound to `f_closure`. It is a lexical binding, however, so it doesn't shadow the global binding of `v`. If we subsequently evaluate this closure, it returns `20`, the value of the local binding of `v`, even though the global `v` is still bound to `10`.

The heart of this implementation of streams is a pair of functions, `delay` and `force`. `delay` takes an expression as argument and does not evaluate it; instead it takes the unevaluated argument and returns a closure. `force` takes a function closure as argument and uses `funcall` to force its application. These functions are defined:

```
(defmacro delay (exp) '(function (lambda ( ) ,exp)))
(defun force (function-closure)
      (funcall function-closure))
```

`delay` is an example of a Lisp form called a *macro*. We cannot define `delay` using `defun` because all functions so defined evaluate their arguments before executing the body. Macros give us complete control over the evaluation of their arguments. We define macros using the `defmacro` form. When a macro is executed, it does not evaluate its arguments. Instead, it binds the unevaluated s-expressions in the call to the formal parameters and evaluates its body *twice*. The first evaluation is called a *macro-expansion*; the second evaluates the resulting form.

To define the `delay` macro, we introducethe *backquote* or `'`. Backquote prevents evaluation just like a `quote`, except that it allows us to evaluate selectively elements of the backquoted expression. Any element of a backquoted s-expression preceded by a comma is evaluated and its value inserted into the resulting expression. For example, assume we have the call `(delay (+ 2 3))`. The expression `(+ 2 3)` is not evaluated; instead it is bound to the formal parameter, `exp`. When the body of the macro is

evaluated the first time, it returns the backquoted expression with the formal parameter, `exp`, replaced by its value, the unevaluated s-expression `(+ 2 3)`. This produces the expression `(function (lambda () (+ 2 3)))`. This is evaluated again, returning a function closure.

If we later pass this closure to `force`, it will evaluate the expression `(lambda () (+ 2 3))`. This is a function that takes no arguments and whose body evaluates to `5`. Using `force` and `delay`, we can implement streams with delayed evaluation. We rewrite `cons-stream` as a macro that takes two arguments and `cons`es the value of the first onto the delayed evaluation of the second. Thus, the second argument may be a function that will return a stream of any length; it is not evaluated. We define `tail-stream` so that it forces the evaluation of the tail of a stream. These are defined:

```
(defmacro cons-stream (exp stream) '(cons ,exp
   (delay ,stream)))

(defun tail-stream (stream) (force (cdr stream)))
```

We also redefine `combine-streams` as a macro that takes two streams but does not evaluate them. Instead, it uses `delay` to create a closure for the second stream and passes this and the first stream to the function `comb-f`. `comb-f` is similar to our earlier definition of `combine-streams`, except that in the event that the first stream is empty, it forces evaluation of the second stream. If the first stream is not empty, the recursive call to `comb-f` is done using our delayed version of `cons-stream`. This freezes the recursive call in a closure for later evaluation.

```
(defmacro combine-streams (stream1 stream2)
      '(comb-f ,stream1 (delay ,stream2)))
(defun comb-f (stream1 stream2)
      (if (empty-stream-p stream1)
          (force stream2)
      (cons-stream (head-stream stream1)
         (comb-f (tail-stream stream1) stream2))))
```

If we add these definitions to the versions of `head-stream`, `make-empty-stream`, and `empty-stream-p` from Section 16.2, we have a complete stream implementation with delayed evaluation.

We can use these functions to solve our problem of producing the first `n` odd Fibonacci numbers. `fibonacci-stream` returns a stream of all the Fibonacci numbers; note that `fibonacci-stream` is a nonterminating recursive function. Delayed evaluation prevents it from looping forever; it produces the next element only when needed. `filter-odds` takes a stream of integers and eliminates the even elements of the stream. `accumulate` takes a stream and a number `n` and returns a *list* of the first `n` elements of the stream.

```
(defun fibonacci-stream (fibonacci-1 fibonacci-2)
      (cons-stream (+ fibonacci-1 fibonacci-2)
         (fibonacci-stream fibonacci-2
             (+ fibonacci-1 fibonacci-2))))
```

```
(defun filter-odds (stream)
        (cond ((evenp (head-stream stream))
              (filter-odds (tail-stream stream)))
                 (t (cons-stream (head-stream stream)
              (filter-odds (tail-stream stream))))))
(defun accumulate-into-list (n stream)
         (cond ((zerop n) nil)
                  (t (cons (head-stream stream)
                         (accumulate-into-list (– n 1)
                              (tail-stream stream))))))
```

To obtain a list of the first 25 odd Fibonacci numbers, we call `accumulate-into-list`:

```
(accumulate-into-list 25
         (filter-odds (fibonacci-stream 0 1)))
```

We may use these stream functions in the definition of the logic programming interpreter of Section 16.3 to improve its efficiency under certain circumstances. Assume that we would like to modify `print-solutions` so that instead of printing all solutions to a goal, it prints the first and waits for the user to ask for the additional solutions. Using our implementation of lists as streams, the algorithm would still search for all solutions before it could print out the first. Using delayed evaluation, the first solution will be the head of a stream, and the function evaluations necessary to find the additional solutions will be frozen in the tail of the stream.

In the next section we modify this logic programming interpreter to implement a Lisp-based expert system shell called `lisp-shell`. Before presenting the expert system shell, however, we mention two additional stream functions that are used in its implementation. In Section 16.3, we presented a general mapping function and a general filter for lists. These functions, `map-simple` and `filter`, can be modified to function on streams. We use `filter-stream` and `map-stream` in the next section; their implementation is an exercise.

17.2 An Expert System Shell in Lisp

The expert system shell developed in this section is an extension of the backward-chaining engine of Section 16.3. The major modifications include the use of certainty factors to manage uncertain reasoning, the ability to ask the user for unknown facts, and the use of a working memory to save user responses. This expert system shell is called `lisp-shell`.

Implementing Certainty Factors

The logic programming interpreter returned a stream of the substitution sets under which a goal logically followed from a database of logical assertions. Bindings that did not allow the goal to be satisfied using the knowledge base were either filtered from the stream or not generated in the first place. In implementing reasoning with certainty factors, however, logic truth values (`t`, `f`) are replaced by a numeric value between `–1` and `1`.

This replacement requires that the stream of solutions to a goal not only contain the variable bindings that allow the goal to be satisfied; they must also include measures of the confidence under which each solution follows from the knowledge base. Consequently, instead of processing streams of substitution sets, `lisp-shell` processes streams of pairs: a set of substitutions and a number representing the confidence in the truth of the goal under those variable substitutions.

We implement stream elements as an abstract data type: the functions for manipulating the substitution and certainty factor pairs are `subst-record`, which constructs a pair from a set of substitutions and a certainty factor; `subst-list`, which returns the set of bindings from a pair; and `subst-cf`, which returns the certainty factor.

Internally, records are represented as dotted pairs, of the form `(<substitution list>. <cf>)`. We next create functions that handle these pairs, the first returning a list of bindings, the second returning a certainty factor, and the third creating substitution-set certainty-factor pairs:

```
(defun subst-list (substitutions)
    (car substitutions))
(defun subst-cf (substitutions)
    (cdr substitutions))
(defun subst-record (substitutions cf)
    (cons substitutions cf))
```

Similarly, rules and facts are stored in the knowledge base with an attached certainty factor. Facts are represented as dotted pairs, `(<assertion>. <cf>)`, where `<assertion>` is a positive literal and `<cf>` is its certainty measure. Rules are in the format `(rule if <premise> then <conclusion> <cf>)`, where `<cf>` is the certainty factor. We next create a sample rule for the domain of recognizing different types of flowers:

```
(rule if (and (rose (var x)) (color (var x) red))
     then (kind (var x) american-beauty) 1)
```

The functions for handling rules and facts are:

```
(defun premise (rule)
    (nth 2 rule))
(defun conclusion (rule)
    (nth 4 rule))
(defun rule-cf (rule)
    (nth 5 rule))
(defun rulep (pattern)
    (and (listp pattern)
         (equal (nth 0 pattern) 'rule)))
```

```
(defun fact-pattern (fact)
     (car fact))
(defun fact-cf (fact)
     (cdr fact))
```

Using these functions, we implement the balance of the rule interpreter through a series of modifications to the logic programming interpreter first presented in Section 16.3.

Architecture of lisp-shell

`solve` is the heart of lisp-shell. `solve` does not return a solution stream directly but first passes it through a filter that eliminates any substitutions whose certainty factor is less than 0.2. This prunes results that lack sufficient confidence.

```
(defun solve (goal substitutions)
   (filter-stream
      (if (conjunctive-goal-p goal)
         (filter-through-conj-goals
            (cdr (body goal))
            (solve (car (body goal))
                 substitutions))
           (solve-simple-goal goal
                 substitutions))
            # '(lambda (x)
            (< 0.2 (subst-cf x)))))
```

This definition of `solve` has changed only slightly from the definition of `solve` in `logic-shell`. It is still a conditional statement that distinguishes between conjunctive goals and simple goals. One difference is the use of the general filter `filter-stream` to prune any solution whose certainty factor falls below a certain value. This test is passed as a lambda expression that checks whether or not the certainty factor of a substitution set/cf pair is less than 0.2. The other difference is to use `solve-simple-goal` in place of infer. Handling simple goals is complicated by the ability to ask for user information. We define `solve-simple-goal` as:

```
(defun solve-simple-goal (goal substitutions)
     (declare (special *assertions*))
     (declare (special *case-specific-data*))
     (or (told goal substitutions
              *case-specific-data*)
      (infer goal substitutions *assertions*)
      (ask-for goal substitutions)))
```

`solve-simple-goal` uses an or form to try three different solution strategies in order. First it calls told to check whether the goal has already been solved by the user in response to a previous query.

User responses are bound to the global variable `*case-specific-data*`; `told` searches this list to try to find a match for the goal. This keeps `lisp-shell` from asking for the same piece of information twice. If this fails, the information was not asked for earlier, and `solve-simple-goal` attempts to infer the goal using the rules in `*assertions*`. Finally, if these fail, it calls `ask-for` to query the user for the information. These functions are defined below.

The top-level read-solve-print loop has changed little, except for the inclusion of a statement initializing `*case-specific-data*` to `nil` before solving a new goal. Note that when `solve` is called initially, it is not just passed the empty substitution set, but a pair consisting of the empty substitution set and a certainty factor of 0. This certainty value has no real meaning: it is included for syntactic reasons until a meaningful substitution set and certainty factor pair is generated by user input or by a fact in the knowledge base.

```
(defun lisp-shell ()
     (declare (special *case-specific-data*))
     (setq *case-specific-data* ( ))
     (prin1 'lisp-shell> )
     (let ((goal (read)))
     (terpri)
     (cond ((equal goal 'quit) 'bye)
           (t (print-solutions goal
                  (solve goal
                     (subst-record nil 0)))
             (terpri)
             (lisp-shell)))))
```

`filter-through-conj-goals` is not changed, but `filter-through-goal` must compute the certainty factor for a conjunctive expression as the minimum of the certainties of the conjuncts. To do so, it binds the first element of `substitution-stream` to the symbol `subs` in a `let` block. It then calls `solve` on the goal and this substitution set; passing the result through the general mapping function, `map-stream`, which takes the stream of substitution pairs returned by `solve` and recomputes their certainty factors as the minimum of the certainty factor of the result and the certainty factor of the initial substitution set. These functions are defined:

```
(defun filter-through-conj-goals (goals
          substitution-stream)
     (if (null goals)
     substitution-stream
     (filter-through-conj-goals (cdr goals)
         (filter-through-goal (car goals)
                substitution-stream))))
```

```
(defun filter-through-goal (goal
                     substitution-stream)
   (if (empty-stream-p substitution-stream)
      (make-empty-stream)
      (let ((subs (head-stream
                 substitution-stream)))
         (combine-streams
         (map-stream (solve goal subs)
              # '(lambda (x)
               (subst-record (subst-list x)
                        (min (subst-cf x)
                    (subst-cf subs)))))
         (filter-through-goal goal
                  (tail-stream
                 substitution-stream))))))
```

The definition of **infer** has been changed to take certainty factors into account. Although its overall structure reflects the version of **infer** written for the logic programming interpreter in Section 16.2, we must now compute the certainty factor for solutions to the goal from the certainty factors of the rule and the certainties of solutions to the rule premise. **solve-rule** calls **solve** to find all solutions to the premise and uses **map-stream** to compute the resulting certainties for the rule conclusion.

```
(defun infer (goal substitutions kb)
   (if (null kb)
      (make-empty-stream)
      (let* ((assertion
         (rename-variables (car kb)))
           (match (if (rulep assertion)
         (unify goal (conclusion assertion)
               subst-list substitutions))
         (unify goal assertion
              (subst-list substitutions)))))
            (if (equal match 'failed)
               (infer goal substitutions
                        (cdr kb))
               (if (rulep assertion)
                  (combine-streams
                     (solve-rule assertion
                       (subst-record match
                       (subst-cf  substitutions)))
                     (infer goal substitutions
                        (cdr kb)))
```

```
                    (cons-stream
                    (subst-record match
                    (fact-cf assertion))
                          (infer goal substitutions
                       (cdr kb))))))))
((defun solve-rule (rule substitutions)
     (map-stream
      (solve (premise rule) substitutions)
        # '(lambda (x) (subst-record
              (subst-list x)
              (* (subst-cf x)
               (rule-cf rule))))))
```

Finally, we modify `print-solutions` to use certainty factors:

```
(defun print-solutions (goal substitution-stream)
   (cond ((empty-stream-p substitution-stream)nil)
         (t (print (apply-substitutions goal
            (subst-list (head-stream
                 substitution-stream))))
           (write-string "cf =")
           (prin1 (subst-cf (head-stream
                    substitution-stream)))
           (terpri)
           (print-solutions goal
                  (tail-stream
              substitution-stream)))))
```

The remaining functions, such as `apply-substitutions` and functions for accessing rules and goals, are unchanged from Section 16.2.

The remainder of `lisp-shell` consists of the functions `ask-for` and `told`, which handle user interactions. These are straightforward, although the reader should note that we have made some simplifying assumptions. In particular, the only response allowed to queries is either "y" or "n". This causes the binding set passed to `ask-for` to be returned with a cf of either 1 or –1, respectively; the user may not give an uncertain response directly to a query. `ask-rec` prints a query and reads the answer, repeating until the answer is either y or n. The reader may expand `ask-rec` to take on any value within the –1 to 1 range. (–1 and 1, of course, offers an arbitrary range; particular applications may use other ranges.)

`askable` verifies whether the user may be asked for a particular goal. Any asked goal must exist as a pattern in the global list `*askables*`; the architect of an expert system may in this way determine which goals may be asked for and which may only be inferred from the knowledge base. `told` searches through the entries in the global `*case-specific-data*` to find whether the user has already answered a query. It is similar to `infer` except it assumes that everything in `*case-specific-data*` is stored as a fact. We define these functions:

```
(defun ask-for (goal substitutions)
  (declare (special *askables*))
  (declare (special *case-specific-data*))
  (if (askable goal *askables*)
      (let* ((query (apply-substitutions goal
                   (subst-list substitutions)))
             (result (ask-rec query)))
  ((setq *case-specific-data*
             (cons (subst-record query result)
                  *case-specific-data*))
          (cons-stream
             (subst-record (subst-list substitutions)
                  result)
             (make-empty-stream)))))

(defun ask-rec (query)
      (prin1 query)
      (write-string ">")
           (let ((answer (read)))
           (cond ((equal answer 'y) 1)
                 ((equal answer 'n) - 1)
                 (t (print
                   "answer must be y or n")
                    (terpri)
                     (ask-rec query)))))

(defun askable (goal askables)
      (cond ((null askables) nil)
         ((not (equal (unify goal car askables) ( ))
                  'failed)) t)
         (t (askable goal (cdr askables)))))

(defun told (goal substitutions case-specific-data)
      (cond ((null case-specific-data)
                   (make-empty-stream))
         (t (combine-streams
             (use-fact goal (car case-specific-data)
                   substitutions)
             (told goal substitutions
                (cdr case-specific-data))))))
```

This completes the implementation of our Lisp-based expert system shell. In the next section we use `lisp-shell` to build a simple classification expert system.

Classification Using lisp-shell

We now present a small expert system for classifying trees and bushes. Although it is far from botanically complete, it illustrates the use and behavior of the `lisp-shell` software. The knowledge base resides in

two global variables: *assertions*, which contains the rules and facts of the knowledge base, and *askables*, which lists the goals that may be asked of the user. The knowledge base used in this example is constructed by two calls to setq:

```
(setq *assertions* '(
   (rule
         if (and (size (var x) tall)
                (woody (var x)))
         then (tree (var x)) .9)
   (rule
         if (and (size (var x) small)
                (woody (var x)))
         then (bush (var x)) .9)
   (rule
         if (and (tree (var x)) (evergreen (var x))
             (color (var x) blue))
         then (kind (var x) spruce) .8)
   (rule
         if (and (tree (var x)) (evergreen (var x))
             (color (var x) green))
         then (kind (var x) pine) .9)
   (rule
         if (and (tree (var x)) (deciduous (var x))
             (bears (var x) fruit))
         then (fruit-tree (var x)) 1)
   (rule
         if (and (fruit-tree (var x))
                (color fruit red)
                (taste fruit sweet))
         then (kind (var x) apple-tree) .9)
   (rule
         if (and (fruit-tree (var x))
                (color fruit yellow)
                (taste fruit sour))
         then (kind (var x) lemon-tree) .8)
   (rule
         if (and (bush (var x))
                (flowering (var x))
                (thorny (var x)))
         then (rose (var x)) 1)
   (rule
         if (and (rose (var x)) (color (var x) red))
         then (kind (var x) american-beauty) 1)))
```

```
(setq *askables* '(
    (size (var x) (var y))
    (woody (var x))
    (soft (var x))
    (color (var x) (var y))
    (evergreen (var x))
    (thorny (var x))
    (deciduous (var x))
    (bears (var x) (var y))
    (taste (var x) (var y))
    (flowering (var x))))
```

A sample run of the trees knowledge base appears below. The reader is encouraged to trace through the rule base to observe the order in which rules are tried, the propagation of certainty factors, and the way in which possibilities are pruned when found to be false:

```
> (lisp-shell)
lisp-shell>(kind tree-1 (var x))
(size tree-1 tall) >y
(woody tree-1) >y
(evergreen tree-1) >y
(color tree-1 blue) >n
(color tree-1 green) >y
(kind tree-1 pine) cf 0.81
(deciduous tree-1) >n
(size tree-1 small) >n
lisp-shell>(kind bush-2 (var x))
(size bush-2 tall) >n
(size bush-2 small) >y
(woody bush-2) >y
(flowering bush-2) >y
(thorny bush-2) >y
(color bush-2 red) >y
(kind bush-2 american-beauty) cf 0.9
lisp-shell>(kind tree-3 (var x))
(size tree-3 tall) >y
(woody tree-3) >y
(evergreen tree-3) >n
(deciduous tree-3) >y
(bears tree-3 fruit) >y
(color fruit red) >n
(color fruit yellow) >y
(taste fruit sour) >y
```

```
(kind tree-3 lemon-tree) cf 0.72
(size tree-3 small) >n
lisp-shell>quit
bye
?
```

In this example, several anomalies may be noted. For example, `lisp-shell` occasionally asks whether a tree is small even though it was told the tree is tall, or it asks whether the tree is deciduous even though the tree is an evergreen. This is typical of the behavior of expert systems. The knowledge base does not know anything about the relationship between tall and small or evergreen and deciduous: they are just patterns to be matched. Because the search is exhaustive, all rules are tried. If a system is to exhibit deeper knowledge than this, these relationships must be coded in the knowledge base. For example, a rule may be written that states that small implies not tall. In this example, `lisp-shell` is not capable of representing these relationships because we have yet to implement the `not` operator. This extension is left as an exercise.

Exercises

1. Rewrite the solution to finding the first `n` odd Fibonacci numbers problem of Section 17.1 so that it uses the general stream filter, `filter-stream`, instead of `filter-odds`. Modify this to return the first `n` even Fibonacci numbers and then modify it again to return the squares of the first `n` Fibonacci numbers.

2. Select a problem such as automotive diagnosis or classifying different species of animals and solve it using `lisp-shell`.

3. Expand the expert system shell of Section 17.2 to allow the user responses other than `y` or `n`. For example, we may want the user to be able to provide bindings for a goal. Hint: This may be done by changing the `ask-for` and related functions to let the user also enter a pattern, which is matched against the goal. If the match succeeds, ask for a certainty factor.

4. Extend `lisp-shell` to include `not`. For an example of how to treat negation using uncertain reasoning, refer to the Prolog-based expert system shell in Chapter 6.

5. In Section 16.3, we presented a general mapping function and a general filter for lists. These functions, `map-simple` and `filter`, can be modified to function on streams. Create the `filter-stream` and `map-stream` functions used in 17.2.

6. Extend `lisp-shell` to produce an answer even when all rules fail to match. In other words, remove the `nil` as a possible result for `lisp-shell`.