MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.188—Supplementary 6.001 course for 6.081 students
Spring Semester, 2007

Project 3

- Issued: March 19
- Due (part 1): April 9
- Due (part 2): April 23

You have a week to complete part 1 of this assignment, and another two weeks to finish part 2. The goal is to give you some experience with pattern-matching and manipulating symbolic data.

1. Building a freshman advisor

In this assignment, you'll give the Institute a hand by building a program that can give academic advice to incoming freshmen. We've sketched a simple framework that you can complete and extend to a more elaborate program. This is an open-ended assignment, involving a lot of list data-structure manipulations. To avoid getting bogged down, it is important that you read through and think about the entire assignment and plan your programming in advance.

Our prototype advisor engages in simple conversations with incoming freshmen. In the following sample conversation, the user's responses are preceded by the prompt symbol **, and the program's responses are shown in upper case. In order to simplify the program, we omit punctuation and represent the responses of both the user and the machine as lists. (That means the user must surround his statements with parentheses.) We also write course numbers using colons instead of periods (e.g., we write 18:03 instead of 18.03).[1]

```
(see-advisor 'hal)
(hi hal)
(i am your freshman advisor)
(what are your plans for the semester)
**(I am going to take 8:01 and 18:01)
(students really enjoy 8:01)
**(I was also thinking of taking 6:081)
(6:081 is too much work for freshmen -- wait until next year)
**(Ok)
(have you considered a context subject)
**(No, but I was thinking about taking a freshman seminar)
(i hear that snorkeling in boston harbor is a really exciting seminar)
**(I think that sounds like fun)
(i am glad to hear that you think that sounds like fun)
**(i was thinking of doing a double major in
   applied partying and business administration)
(applied partying is fascinating and you can make a living doing it
```

[1] If we used a period, Scheme would treat course numbers as decimal numbers rather than as ordinary symbols. We don't want to use numbers here. For instance, we wouldn't want Scheme to "simplify" 18.100 to 18.1.

```
   if business administration does not work out)
**(Goodbye)
(goodbye hal)
(have a good semester!)
```

Although the advisor program seems to understand and reply to the user's remarks, the program in fact ignores most of what the user types and has only three rudimentary methods for generating responses. One method, illustrated above by the exchange

```
**(I think that sounds like fun)
(i am glad to hear that you think that sounds like fun)
```

takes the user's reply, changes some first-person words like "I," "me," "my," and "am" to the corresponding second-person words, and appends the transformed response to a phrase such as "I am glad to hear that" or "you say." The second method used by the program is to completely ignore what the user types and simply respond with some sort of general remark like "Have you considered a context subject," or "MIT has a lot of interesting departments."

The third reply-generating method is the most complicated and flexible. It uses a *pattern matcher*, similar to the one assigned in the tutor problems for this week. We'll discuss the details of this method below.

Overview of the advisor program

Every interactive program, including the Scheme interpreter itself, has a distinguished procedure called a *driver loop*. A driver loop repeatedly accepts input, processes that input, and produces the output. `See-advisor`, the top-level procedure of our program, first greets the user, then asks an initial question and starts the driver loop.

```
(define (see-advisor name)
  (print (list 'hi name))
  (print '(i am your freshman advisor))
  (print '(what are your plans for the semester))
  (advisor-driver-loop name))

(define (advisor-driver-loop name)
  (newline)
  (display '**)
  (let ((user-response (read)))
    (cond ((equal? user-response '(goodbye))
           (print (list 'goodbye name))
           (print '(have a good semester!)))
          (else (print (reply-to user-response))
                (advisor-driver-loop name)))))
```

The driver loop prints a prompt and reads the user's response.[2] If the user says `(goodbye)`, then the program terminates. Otherwise, it calls the following `reply-to` procedure to generate a reply according to one of the methods described above.

```
(define (reply-to input)
  (cond ((generate-match-response conventional-wisdom input))
        ((with-odds 1 2) (reflect-input input))
        (else (pick-random general-advice))))
```

`Reply-to` is implemented as a Lisp `cond`, one `cond` clause for each basic method for generating a reply. The clause uses a feature of `cond` that we haven't seen before—if the `cond` clause consists of a single expression, this serves as both "predicate" and "consequent". Namely, the clause is evaluated and, if the value is not false, this is returned as the result of the `cond`. If the value is false, the interpreter proceeds to the next clause. So the first clauses above works because we have arranged for `generate-match-response` to return false if it has no response to generate.

Notice that the order of the clauses in the `cond` determines the priority of the advisor's response-generating methods. As we have arranged it above, the advisor will reply with a matcher-triggered response whenever there is one. Failing that, the advisor will either (with odds 1 in 2) repeat the input after transforming first person to second person (`reflect-input`) or give an arbitrary piece of general advice. The predicate

```
(define (with-odds n1 n2) (< (random n2) n1))
```

returns true with designated odds (`n1` in `n2`). When you modify the advisor program, feel free to change the priorities of the various methods.

Disbursing random general advice

The advisor's easiest advice method is to simply pick some remark at random from a list of general advice such as:

```
(define general-advice
  '((make sure to take some humanities)
    (have you considered a context subject)
    (mit has a lot of interesting departments)
    (make sure to get time to explore the Boston area)
    (how about a freshman seminar)))
```

`Pick-random` is a useful little procedure that picks an item at random from a list:

```
(define (pick-random list) (list-ref list (random (length list))))
```

[2]This uses the Scheme primitive `read`, which waits for you to type an expression and press return. The value returned by `read-from-keyboard` is the expression you type. Remember that you'll need to surround your input with parentheses if you want `read` to return a list.

Changing person

To change "I" to "you", "am" to "are", and so on, the advisor uses a procedure `sublist`, which takes an input `list` and another list of `replacements`, which is a list of pairs.

```
(define (change-person phrase)
  (sublist '((i you) (me you) (am are) (my your))
           phrase))
```

For each item in the `list` (note the use of `map`), `sublist` substitutes for that item, using the `replacements`. The `substitute` procedure scans down the `replacement` list, looking for a pair whose `car` is the same as the `item`. If so, it returns the `cadr` of that pair. If the list of `replacements` runs out, `substitute` returns the `item` itself.

```
(define (sublist replacements list)
  (map (lambda (elt) (substitute replacements elt))
       list))

(define (substitute replacements item)
  (cond ((null? replacements) item)
        ((eq? item (caar replacements)) (cadar replacements))
        (else (substitute (cdr replacements) item))))
```

The advisor's response method, then, uses `change-person`, gluing a random innocuous beginning phrase (which may be empty) onto the result of the replacement:

```
(define (reflect-input input)
  (append (pick-random beginnings) (change-person input)))

(define beginnings
  '((you say)
    (why do you say)
    (i am glad to hear that)
    ()))
```

Using a pattern matcher

Consider this interaction from our sample dialogue with the advisor:

```
**(i was thinking of doing a double major in
   applied partying and business administration)
(applied partying is fascinating and you can make a living doing it
   if business administration does not work out)
```

The advisor has identified that the input matches a *pattern*:

```
(<stuff> double major in <stuff> and <stuff>)
```

and used the match results to fill in an appropriate *skeleton* for the response.

The pattern matcher used here is similar to the one in the tutor problems, but there are some important differences. The most important difference is that the dummy variables in the pattern can match an arbitrary number (zero or more) consecutive terms. Here is the pattern and skeleton for the example above:

```
pattern:  (* double major in * and *)
skeleton: ((: 1) is fascinating and you can make a living doing it
            if (: 2) does not work out)
```

When the pattern is matched against some test expression, it will return either `failed` if there is no match, or else a list of lists—one for each star in the pattern—saying what sequence the star matched against. For example,

```
==> (match '(* double major in * and *)
           '(i was thinking of doing a double major in
             applied partying and business administration))

 ((i was thinking of doing a)
  (applied partying)
  (business administration))
```

The first list in the match result contains what the first star in the pattern matched against, and so on. When the skeleton is *instantiate-entry* against some match result, the value `(: 0)` will be filled in with the first list, `(: 1)` with the second list, and so on:[3]

```
==>(instantiate-entry
    '((: 1) is fascinating and you can make
       a living doing it if (: 2) does not work out)
    '((i was thinking of doing a)
      (applied partying)
      (business administration)))

(applied partying is fascinating and you can make a living
  doing it if business administration does not work out)
```

Here is the advisor's table of conventional wisdom. The table is represented as a list of *entries*, each consisting of a pattern and a skeleton:

```
(define (make-entry pattern skeleton)
  (list pattern skeleton))

(define (entry-pattern  x) (car  x))
(define (entry-skeleton x) (cadr x))
```

[3]Note that `instantiate-entry` is not defined in the appended code — you will have to implement it as part of the first week's homework.

```
(define conventional-wisdom
  (list
   (make-entry
    '(* 6:081 *)
    '(6:081 is too much work for freshmen -- wait until next year))
   (make-entry
    '(* 8:01 *)
    '(students really enjoy 8:01))
   (make-entry
    '(* seminar *)
    '(i hear that snorkeling in Boston Harbor is a really
      exciting seminar))
   (make-entry
    '(* want to take * next *)
    '(too bad -- (: 1) is not offered next (: 2)))
   (make-entry
    '(* double major in * and *)
    '((: 1) is fascinating and you can make a living
       doing it if (: 2) does not work out))
   (make-entry
    '(* double major *)
    '(doing a double major is a lot of work))
   ))
```

Here is the procedure that matches an input against some item in the table, and returns the instantiate-entry skeleton (or returns `failed`).

```
(define (match-to-table-entry entry input)
  (let ((match-result (match (entry-pattern entry) input)))
    (if (good-match? match-result)
        (instantiate-entry (entry-skeleton entry) match-result)
        'failed)))
```

Note from the table that some inputs might match more than one pattern. When this happens the advisor generates all possible responses and then picks one at random to print. We'll leave you to implement the procedure `generate-match-response` that accomplishes this (problem 3.4 below).

Details of the matcher

The matcher simply takes a pattern and an expression and returns the list showing how the stars (arbitrary segments) matched.

```
(define (match pat exp)
  (cond ((and (null? pat) (null? exp)) '())
        ((null? pat) 'failed)
        ((start-arbitrary-segment? pat) (match-arbitrary-segment pat exp))
```

```
((null? exp) 'failed)
((equal? (car pat) (car exp)) (match (cdr pat) (cdr exp)))
(else 'failed)))
```

The only tricky part of the matcher is in the handling of arbitrary segments. There are three possibilities for a match when the pattern begins with a star.

1. If the rest of the pattern successfully matches the rest of the expression, then the star should match the first item in the expression. For example, given the pattern `(* is a quail)` and the expression `(dan is a quail)` then there will be a match with the star matching `(dan)`.

2. If the entire pattern (recursively) successfully matches the rest of the expression, then we take whatever the star matched against in that recursive match and *extend* it by adjoining the first item in the expression; e.g., given the pattern `(* is a quail)` and the expression `(handsome dan is a quail)`, we recursively match the pattern against the rest of the expression (with the star matching against `(dan)`) and then extend this to by adjoining `handsome` to produce a successful match with star matching `(handsome dan)`. This step happens recursively, and it will keep recursing as long as the successive rests of the expression match. For example, the same pattern will match `(strong noble handsome dan is a quail)`.

3. If the rest of the pattern matches the entire expression, then we produce a match with the initial star matching the empty list.

Note that possibility 3 is the only one available if the expression is empty.

This algorithm is implemented by the following procedures:

```
(define (match-arbitrary-segment pat exp)
  (if (null? exp)
      (match-rest-pat pat exp)
      (let ((first-try (match-rest-rest pat exp)))
        (if (good-match? first-try)
            first-try
            (let ((second-try (match-rest-exp pat exp)))
              (if (good-match? second-try)
                  second-try
                  (match-rest-pat pat exp)))))))

(define (match-rest-pat pat exp)
  (let ((result (match (cdr pat) exp)))
    (if (good-match? result)
        (extend-dictionary '() result)
        'failed)))

(define (match-rest-exp pat exp)
  (let ((result (match pat (cdr exp))))
    (if (good-match? result)
        (extend-first-entry (car exp) result)
        'failed)))
```

```
(define (match-rest-rest pat exp)
  (let ((result (match (cdr pat) (cdr exp))))
    (if (good-match? result)
        (extend-dictionary (list (car exp)) result)
        'failed)))
```

There are also some simple utilities:

```
(define (good-match? m)
  (not (eq? m 'failed)))

(define (start-arbitrary-segment? pat)
  (eq? (car pat) '*))

(define (extend-dictionary new-entry matches)
  (cons  new-entry  matches))

(define (extend-first-entry extension matches)
  (cons (cons extension (car matches)) (cdr matches)))
```

The matcher is a rather complex example of recursion. You might not feel that you could design such a program from scratch, but you ought to understand how this one works.

A complete listing of the advisor program, almost all of which was described above, is attached to this problem set.

2. To do for April 9

Begin by loading the code, and try it out by typing

```
(see-advisor '<your name>)
```

In this version you load, the advisor will never attempt to use the pattern matcher (because the procedure `generate-match-response` has been "dummied out" to always return `#f`), so all you will obtain are the simple "you-me" transformations and the random general advice.

Problem 1 Expand the advisor's store of conventional wisdom and general advice. You may find it helpful to consult MIT course evaluation guides—or fabricate your own helpful advice. Try out the modified program. You needn't turn in anything for this problem.

Problem 2 The pattern matcher loaded with the code is missing the `instantiate-entry` procedure. Write this. `Instantiate-entry` should take a skeleton and a list of lists (showing what the pattern stars were bound to) and return the skeleton with the appropriate pieces spliced in. Here are some examples:

```
==>(instantiate-entry
    '((: 1) was (: 0) on the (: 2))
    '((shining) (the sun) (sea)))
(the sun was shining on the sea)

==>(instantiate-entry
    '(my (: 0) my (: 0) my (: 1) a (: 0))
    '((horse) (kingdom for)))
(my horse my horse my kingdom for a horse)
```

Turn in a listing of your procedure together with some examples demonstrating that it works.

2. To do for April 23

Problem 3 Now that `instantiate-entry` is working, you should be able to call the procedure `match-to-table-entry` (already loaded with the code and described above). Use `make-entry` to define a sample pattern-skeleton entry and demonstrate that `match-to-table-entry` works with a few sample inputs.

Problem 4 Now implement the procedure `generate-match-response`. Replace the dummy definition (loaded with the file) by your new definition. Your procedure should take a table of entries and an input, match the input to each table entry, and pick at random one of the non-failure responses. If there are no non-failure responses, the procedure should return `#f`. If you think about this procedure carefully, you will realize that is a simple combination of `map`, `filter` (see exercise 2.3 above), and `pick-random`. You'll have to define `filter` if you want to use it. Turn in a listing of the procedure(s) you write here together with some test examples to show that they work. You can use the advisor's predefined `conventional-wisdom` table for your tests.

Problem 5 Now the entire advisor program should work, including the responses generated by the matcher. Try this to check. Add some new entries to the `conventional-wisdom` table. Turn in a few of the new entries you added, together with some sample responses showing the advisor in operation.

Problem 6 Design and implement some other improvement that extends the advisor‘s capabilities.

For example, the program currently uses no information about the student. Instead of calling `see-advisor` with only the student's name, you might invent a student-record data structure that the advisor could consult. Alternatively, you might give the advisor some sort of "memory" or "context" so that future responses are affected by past responses. As a third idea, you might use the matcher in a different way. For example, instead of just instantiating a skeleton with a match result, you could pass the result to a procedure for more flexible treatment.

Implement your modification. You need not feel constrained to follow the suggestions given above. You needn't even feel constrained to implement a freshman advisor. Perhaps there are other

members of the MIT community who you think could be usefully replaced by simple programs: a dean or two, *The Tech*, the Registrar, a 6.081 lecturer, ...

Turn in descriptions (in English) of the main procedures and data structures used, together with listings of the procedures and a sample dialogue showing the program in operation.

This problem is not meant to be a major project. Don't feel that you have to do something elaborate.