

University of Massachusetts Lowell

91.301: Organization of Programming Languages
Fall 2004

Problem Set 5 **Freshman advisor**

Issued: Section 201 (Dimock): Friday, 8 Oct 2004; Section 202 (Yanco): Thursday, 7 Oct 2004
Due: Section 201: Friday, 15 Oct 2004; Section 202: Thursday, 14 Oct 2004
Reading: Text (SICP 2nd Edition by Abelson & Sussman): Sections 2.2 and 2.3

Overview

In this problem set, you'll work with lists and symbols to create a freshman advisor.
Use the DrScheme language "Textual (MzScheme, uses R5RS)" for this problem set.

What to turn in

Keep your answers in a file named `ps5-ans.ss` (download the start of this file from the course web site). Put the code for each problem sequentially in the definitions buffer. (If you work problems in a different sequence, put them in order before you submit.) Put the problem number in a comment before the code for that problem (use a semi-colon to make a line a comment). Underneath the code for each problem, cut and paste the appropriate sample runs. Please put semi-colons in front of the sample runs, which will comment them out and allow you to load the entire buffer in another session.

Print the definitions buffer (or the file `ps5-ans.ss`, which should be equivalent) and turn it in during the class on the due date. Also submit the answers using the submit command:

Section 201: `submit dimock ps5 ps5-ans.ss`

Section 202: `submit holly 301-ps5 ps5-ans.ss`

1. Building a freshman advisor

In this assignment, you'll give aid your CS professors 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.¹ 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 (the program won't actually output uppercase). In order to simplify the program, we omit punctuation and represent the responses of both the user and the machine as lists. We also write course numbers using colons instead of periods (e.g., we write 91:301 instead of 91.301).²

```
(see-advisor 'holly)
(HI HOLLY)
(I AM YOUR FRESHMAN ADVISOR)
(WHAT ARE YOUR PLANS FOR THE SEMESTER)
** (I have already planned my schedule)
(I AM GLAD TO HEAR THAT YOU HAVE ALREADY PLANNED YOUR SCHEDULE)
** (nothing fits)
(UML HAS A LOT OF INTERESTING DEPARTMENTS BUT NO PARKING)
** (what is 91:102 about)
(91:102 IS ABOUT FUN WITH POINTERS)
** (can I take 95:142)
(THE PREREQUISITES FOR 95:142 ARE 95:141)
** (i want to take 91:301)
(91:301 IS TOO MUCH WORK FOR FRESHMEN -- WAIT UNTIL NEXT YEAR)
** (what are 91:101 95:141 and 92:132 about)
(91:101 IS ABOUT GETTING TO KNOW UNIX)
(95:141 IS ABOUT INTRO PHYSICS)
(92:132 IS ABOUT DOING MORE CALCULUS)
** (how many credits is 91:101)
(91:101 IS A 4 CREDIT SUBJECT)
** (i want to double major in math and stamp collecting)
(MATH IS FASCINATING AND YOU CAN MAKE A LIVING DOING IT
 IF STAMP COLLECTING DOES NOT WORK OUT)
** (goodbye)
(GOODBYE HOLLY)
(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 four rudimentary methods for generating responses. One method, illustrated above by the exchange

```
** (I have already planned my schedule)
(I AM GLAD TO HEAR THAT YOU HAVE ALREADY PLANNED YOUR SCHEDULE)
```

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 "UML has a lot of interesting departments but no parking."

The other two reply-generating methods are more complicated and flexible. They use a *pattern matcher*. One method uses the pattern matcher alone. The other uses the matcher in conjunction with a database drawn from the UML catalog.

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.

¹This advisor program is based on the famous Eliza program written in the 1960's by MIT professor Joseph Weizenbaum. Eliza carried on "conversations" similar to those in non-directive psychiatric analysis.

²If 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" a course numbered 18.100 to 18.1.

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

(define (advisor-driver-loop name)
  (let ((user-response (prompt-for-command-expression "** ")))
    (cond ((equal? user-response 'goodbye)
           (write-line (list 'goodbye name))
           (write-line '(have a good semester!)))
          (else (reply-to user-response)
                 (advisor-driver-loop name))))))
```

The driver loop prints a prompt and reads the user's response. 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
   ((translate-and-run input subject-knowledge))
   ((translate-and-run input conventional-wisdom))
   ((with-odds 1 2)
    (write-line (reflect-input input)))
   (else
    (write-line (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, for example, the first clause above works because we have arranged for `translate-and-run` to return false if it does not generate a response.

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 if possible with a matcher-triggered response based on `subject-knowledge`. Failing that, the advisor attempts to generate a response based upon its repertoire of conventional wisdom. 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
  '((be sure to take some gen eds)
    (uml has a lot of interesting departments but no parking)
    (try to catch a spinners game)
    (how about a robotics class)
    (beware of the university ave bridge)))
```

`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))))
```

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 two-element lists.

```
(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 the `replacements` list, looking for a replacement whose first element is the same as the given item. If so, it returns the second element of that replacement. (Note the use of `caar` and `cadar` since `replacements` is a list of lists.) 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)
    ()))
```

Pattern matching and rules

Consider this interaction from our sample dialogue with the advisor:

```
** (i want to double major in math and stamp collecting)
(MATH IS FASCINATING AND YOU CAN MAKE A LIVING DOING IT
 IF STAMP COLLECTING 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 result to trigger an appropriate procedure to generate the response. Each rule consists of a *rule pattern* and a *rule action* procedure that is run if the pattern matches the input data.

Patterns consist of *constants* (that must be matched literally) and *pattern variables*. There are two kinds of pattern variables. A variable designated by a question mark matches a single item. A variable designated by double question marks matches a list of zero or more items. Thus, the pattern

```
((? x) double major in (?? y) and (?? z))
```

matches

```
(i want to double major in math and stamp collecting)
```

with *x* matching the list (i want to) and *y* matching the list (math) and *z* matching the list (stamp collecting).³

The pattern matcher produces a *dictionary* that associates each pattern variable to the value that it matches. The *action* part of a rule is a procedure that takes this dictionary as argument and performs some action (e.g., printing the advisor's response). In the case of our double major example, we lookup the values associated to *y* and *z* in the dictionary and print an appropriate response. For this rule, the action procedure is

```
(lambda (dict)
  (write-line
   (append
    (value 'y dict)
    '(is fascinating and you can make a living doing it if)
    (value 'z dict)
    (does not work out))))
```

Here is the advisor's complete repertoire of conventional wisdom rules:

```
(define conventional-wisdom
  (list
   (make-rule
    '((?? x) 91:301 (?? y))
    (simple-response '(91:301 is too much work for freshmen
-- wait until junior year)))
   (make-rule
    '((?? x) 91:450 (?? y))
    (simple-response '(students really enjoy 91:450)))
   (make-rule
    '((?? x) seminar (?? y))
    (simple-response '(i hear that snorkeling in the Merrimack River
is a really exciting seminar)))
   (make-rule
    '((?? x) to take (?? y) next (?? z))
    (lambda (dict)
      (write-line
       (append '(too bad -- )
                (value 'y dict)
                '(is not offered next)
                (value 'z dict)))))
   (make-rule
    '((?? x) double major in (?? y) and (?? z))
    (lambda (dict)
      (write-line
       (append
        (value 'y dict)
        '(is fascinating and you can make a living doing it if)
        (value 'z dict)
        '(does not work out))))))
   (make-rule
    '((?? x) double major (?? y))
    (simple-response '(doing a double major is a lot of work)))
  ))
```

Notice that for some of these rules, the action procedure has a particularly simple form that ignores the dictionary and just prints a constant response:

```
(define (simple-response text)
  (lambda (dict) (write-line text)))
```

The clause in the `reply-to` procedure that checks these rules is

```
(translate-and-run input conventional-wisdom)
```

³Notice that *y* is bound to the list (math), not the symbol math. We could demand that the matches to *y* and *z* be single items by writing the pattern as (?? x) double major in (? y) and (? z). This would match (I want to double major in math and baseball), but it would not match (I want to double major in math and stamp collecting).

`translate-and-run` takes a list of pattern-action rules and runs the rules on the input. If any of the rules is found applicable, `translate-and-run` returns true, otherwise it returns false. This is implemented in terms of the procedure `try-rules`, which invokes the matcher with appropriate procedures to be executed if the match succeeds and fails.

```
(define (translate-and-run input rules)
  (try-rules input rules
    (lambda () false)           ;fail
    (lambda (result fail) true))) ;succeed
```

Using catalog knowledge

An interchange such as

```
** (what is 91:102 about)
(91:102 IS ABOUT FUN WITH POINTERS)
** (can I take 95:142)
(THE PREREQUISITES FOR 95:142 ARE 95:141)
```

requires actual knowledge of UML subjects. This is embodied in the advisor's "catalog":

```
(define catalog
  (list
    (make-entry '91:101 'cs '(getting to know unix) 4 '(cs) '())
    (make-entry '91:102 'cs '(fun with pointers) 4 '(cs) '(91:101))
    (make-entry '91:201 'cs '(objects) 4 '(cs) '(91:102))
    (make-entry '91:203 'cs '(assembly) 4 '(cs) '(91:102))
    (make-entry '91:204 'cs '(writing programs) 3 '(cs) '(91:201))
    (make-entry '91:301 'cs '(fun with scheme) 3 '(cs) '(91:204))
    (make-entry '95:141 'physics '(intro physics) 4 '(lab) '())
    (make-entry '95:142 'physics '(more physics) 4 '(lab) '(95:141))
    (make-entry '92:131 'math '(learning calculus) 4 '(math) '())
    (make-entry '92:132 'math '(doing more calculus) 4 '(math) '(92:131))
  ))
```

Each entry in this list provides information about a subject as a simple list data structure:

```
(define (make-entry subject department summary units satisfies prerequisites)
  (list subject department summary units satisfies prerequisites))

(define (entry-subject entry) (list-ref entry 0))
(define (entry-department entry) (list-ref entry 1))
(define (entry-summary entry) (list-ref entry 2))
(define (entry-units entry) (list-ref entry 3))
(define (entry-satisfies entry) (list-ref entry 4))
(define (entry-prerequisites entry) (list-ref entry 5))
```

This knowledge permits the advisor to use rules like the following:

```
(make-rule
  '(can I take (? s ,in-catalog))
  (lambda (dict)
    (let ((entry (value 's dict)))
      (write-line
        (append '(the prerequisites for)
          (list (entry-subject entry))
          '(are)
          (entry-prerequisites entry)))))) )
```

Notice that the pattern here specifies that the variable `s` satisfies the *restriction* defined by `in-catalog`:

```
(define (in-catalog subject fail succeed)
  (let ((entry (find subject catalog)))
    (if entry
      (succeed entry fail)
      (fail))))
```

Restrictions that interface to the pattern matcher take as arguments procedures that should be called depending on whether the restriction succeeds or fails. `in-catalog` succeeds if the `subject` actually names a subject in the catalog. The value passed to the `succeed` procedure, namely, the catalog entry, will be the value associated to the pattern variable in the dictionary.⁴

See the `adv.ss` file for the complete list of subject knowledge rules. Notice that some of the rules use the restriction `subjects` that match either a single subject in the catalog or a sequence “ $\langle s_1 \rangle \langle s_2 \rangle \dots$ and $\langle s_n \rangle$ ” where the s ’s are in the catalog. See the attached code for how this restriction is implemented.

2. Warm-Up Exercises

Some preliminary practice with these ideas should prove extremely helpful in doing the exercises in the next section. Turn in the answers to these questions with your problem set.

Three useful procedures

The code for the problem set contains the following procedures, which are generally useful in working with lists:

```
(define (list-union l1 l2)
  (cond ((null? l1) l2)
        ((member (car l1) l2)
         (list-union (cdr l1) l2))
        (else
         (cons (car l1)
               (list-union (cdr l1) l2)))))

(define (list-intersection l1 l2)
  (cond ((null? l1) '())
        ((member (car l1) l2)
         (cons (car l1)
               (list-intersection (cdr l1) l2)))
        (else (list-intersection (cdr l1) l2))))

(define (reduce combiner initial-value list)
  (define (loop list)
    (if (null? list)
        initial-value
        (combiner (car list) (loop (cdr list)))))
  (loop list))
```

`List-union` takes two lists and returns a list that contains the (set-theoretic) union of the elements in the lists.⁵ `List-intersection`, similarly returns the intersection.

Exercise 1: What is the difference between `list-union` and `append`? Given an example where they return the same result; where they return a different result.

Exercise 2: `Reduce` takes a procedure that combines two elements, together with a list and an initial value, and uses the combiner to combine together all the elements in the list, starting from the initial value. For example,

⁴Note, by the way, that the rule definition uses backquote (‘) and comma in order to piece together a list that includes both literal symbols and the *value* of the procedure `in-catalog`.

⁵That is, the resulting list will have no duplicate elements, assuming that the argument lists have no duplicates.

```
(reduce + 0 '(1 2 3 4 5 6 7 8 9 10))
```

returns 55.

Show how to use `reduce` to compute the product of all the elements in a list of numbers.

Exercise 3: Use `map` together with `reduce` to compute the sum of the squares of the elements in a list of numbers.

Exercise 4: What result is returned by

```
(reduce append '() '((1 2) (2 3) (4 6) (5 4)))
```

How would the result be different if we used `list-union` instead of `append`?

Exercise 5: What does the following procedure do, given a list of symbols? What would be a more descriptive name for the procedure?

```
(define (proc symbols)
  (reduce list-union
    '()
    (map list symbols)))
```

3. The Advisor

Begin by opening the `ps5-ans.ss` file and executing it. This file will load the code for problem set 5: `utils.ss`, `match.ss` and `adv.ss` (all available on the course web site).

Start the advisor program by typing

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

and try some sample questions. Make sure that your questions are in parentheses (are lists).

Exercise 6: Expand the advisor's store of `beginnings` and `general-advice`, either from real experience or fabricated. Turn in a short demonstration transcript of the conversation.

Exercise 7: For each of the rules in `conventional-wisdom` and `subject-knowledge`, type an input that will trigger this rule. Examine each of the rule action procedures that gets invoked and make sure that you understand what they are doing and how they generate the advisor's response. Turn in the list of sentences, one for each rule, in the order in which the rules appear in the code. For each sentence, indicate on your transcript the input fragments that match the pattern variables in the relevant rule.

Exercise 8: The `entry-prerequisites` procedure used by the advisor returns the list of prerequisite subjects as listed in the catalog. For example, the prerequisite for 91:301 is 91:204. But certainly to take 91:301 one must also take the prerequisites of the prerequisites, and the prerequisites of these, ad nauseum. Implement a program called `all-prerequisites` that finds (recursively) all the real prerequisites of a given subject and returns a list in which each item appears only once. If the "subject" does not appear in the catalog, assume it has no prerequisites. (You may find it useful to use the procedure `list-union` included in the code.) Demonstrate that your procedure computes all the prerequisites of 91:301 to be (91:204 91:201 91:102 91:101) (although not necessarily listed in that order; the order will depend upon your code). (Hint: There are a lot of ways to do this, and many of them lead to complex recursive programs. See if you can find a simple way to express the solution in terms of `list-union` and `reduce`.)

Exercise 9: Using `all-prerequisites`, add a new rule to `subject-knowledge` that answers questions of the form “Can I take $\langle subject_1 \rangle$ if I have not taken $\langle subject_2 \rangle$?” Turn in your rule and a sample showing that it works.

Exercise 10: Write a procedure `total-credits` that takes a list of subjects and returns the total number of credits of all the subjects in the list. For example, if the list is (91:102 95:142 92:132) the total is 12 credits. Turn in your code and an example that shows your procedure works.

Lab Exercise 11: Design and implement some other improvement that extends the advisor’s capabilities.

For example, there is not much use of the subject summary information. For example, you might ask the advisor if there is a subject about electricity. You could add more entries to the catalog or include more information in the catalog entries.

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 UML community who you think could be usefully replaced by simple programs: a dean or two, the Registrar, your 91.301 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.

Contest

Prizes will be awarded for the cleverest programs and dialogues turned in for this problem set.