Structure and Interpretation of Computer Programs
October 2000

Problem Set 5
Freshman Advisor

Issued: Wednesday, 11 October 2000
Due: Friday, 13 October 2000
Reading: Text (SICP 2nd Edition by Abelson & Sussman): Section 2.3

Overview

Burgundy University, not too far from arsDigita University, has decided to eliminate all freshman advisors and replace them with computer terminals running, "you know, some kind of AI kind of type thingy." When they started this project, the Burgundy Administration thought it would be a good term assignment for one of their graduate programming classes. Thus, Dean Travis Teeh, head of the effort, gave it to Professor Snoothead to use in his course, "Advanced Programming Techniques with Baroque Languages." Unfortunately, not a single one of Snoothead's crack students was able to complete a functioning system.

So, Dean Teeh decided to outsource the project and contacted arsDigita to hire them on at $200/hr. Well, things are a little busy right now up at the corporate site, so the problem has been transferred down to us at the University to use as a problem set in our programming course. Can you achieve in a few days what Burgundy graduate students were unable to complete in a full term?[1]

1. Building a Freshman Advisor

In this assignment, you will give Burgundy University a hand by building a program that can give course selection advice to incoming freshmen. We have 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 students.[2] In the following sample conversation, the user's responses are preceded by the prompt symbol **, and the program's responses are shown in slanted font (although they will not appear as such on your computers). In order to simplify the program, we omit punctuation and represent the responses of both the user and the machine as lists.

[1] This problem set is strongly based on the "Freshman Advisor" problem set on the MIT Press SICP web site.

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

Course numbers are concatenations of the department and the level, as is common in many universities. Courses are generally organized into introductory, secondary, and advanced levels, corresponding to numbers in the 100, 200, and 300 range. As the abbreviation for the Biology department is BIO, the introductory freshman biology course (known in the vernacular as "Squishy Things") is called BIO101; a secondary-level organic chemistry course ("Fun with Benzene") is CHEM224; an advanced course in mathematical analysis ("Hard Equations") is MATH319. Most, but not all, courses have some kind of prerequisite, and some fulfill various distribution and concentration requirements.

```
(see-advisor 'ben)
(hi ben)
(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)
(burgundy has a lot of interesting departments)
** (what is geo104 about)
(geo104 is about planets)
** (can I take geo104)
(the prerequisites for geo104 are math203 phys102)
** (can I take math203 and eecs101)
(eecs101 is too much work for freshmen -- wait until next year)
** (what are chem111 bio212 and math102 about)
(chem111 is about smelly organic crud and goop)
(bio212 is about diseases and their applications)
(math102 is about multivariate calculus)
** (what are the prerequisites for math203)
(the prerequisites for math203 are math102)
** (how many credits is math102)
(math102 is a 4 credit subject)
** (can I take 23 credits this term)
(make sure to get time to explore the Boston area)
** (is there a good seminar)
(i hear that snorkeling in Boston harbor is a really exciting seminar)
** (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 ben)
(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 "Burgundy has a lot of interesting departments."

The other two reply-generating methods are more complicated and flexible. They use a *pattern matcher*, similar to the one presented in class. One method uses the pattern matcher alone. The other uses the matcher in conjunction with a database drawn from the Burgundy University 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.

```
(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)
  #t
  )

(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.[3] If the user says `(goodbye)`, then the program terminates and returns true. 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 Scheme `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:

[3] This uses the Scheme primitive `prompt-for-command-expression`, which prints the specified string as a prompt and waits for you to type an expression, followed by `C-x C-e`. The value returned by `prompt-for-command-expression` is the expression you type.

```
(define general-advice
  '((make sure to take some humanities)
    (burgundy 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))))
```

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.

The rule system used here is similar to the one presented in class. 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)`. Notice 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)`.

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 look up 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) eecs101 (?? y))
              (simple-response '(eecs101 is too much work for freshmen
                                         -- wait until next year)))
   (make-rule '((?? x) phys101 (?? y))
              (simple-response '(students really enjoy phys101)))
   (make-rule '((?? x) seminar (?? y))
              (simple-response '(i hear that snorkeling in Boston Harbor
                                    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)
```

`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, as explained in class, 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 geo104 about)
(geo104 is about planets)
** (can I take geo104)
(the prerequisites for geo104 are math203 phys102)
```

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

```
(define catalog
  (list
   (make-entry 'phys101 'physics   '(classical mechanics) 4
                                   '(gur physics) '())
   (make-entry 'phys102 'physics   '(electricity and magnetism) 4
                                   '(gur physics) '(phys101 math101))
   (make-entry 'phys103 'physics   '(waves) 4
                                    '(rest) '(phys102 math102))
   (make-entry 'phys304 'physics   '(quantum weirdness) 4
                                   '(rest) '(phys103 math203))
   (make-entry 'math101 'math      '(elementary differential and integral calculus) 4
                                   '(gur calculus) '())
   (make-entry 'math102 'math      '(multivariate calculus) 4
                                   '(gur calculus) '(math101))
   (make-entry 'math203 'math      '(differential equations) 4
                                   '(rest) '(math102))
   (make-entry 'math204 'math      '(theory of functions of a complex variable) 4
                                   '() '(math203))
   (make-entry 'math319 'math      '(hard equations) 4
                                   '() '(math204))
   (make-entry 'eecs101 'eecs      '(scheming with yanco and pezaris) 5
                                   '(rest) '(true-grit))
   (make-entry 'eecs202 'eecs      '(circuits) 5
                                   '(rest) '(phys102 math102))
   (make-entry 'chem291 'chemistry '(like crystals dude) 4
                                   '(gur chemistry) '())
   (make-entry 'chem224 'chemistry '(fun with benzene) 4
                                   '(gur chemistry) '(chem111))
   (make-entry 'chem111 'chemistry '(smelly organic crud and goop) 4
                                   '(gur chemistry) '(a-strong-stomach))
   (make-entry 'hist101 'history   '(what has been) 3
                                   '(gur history) '())
   (make-entry 'hist102 'history   '(what will be) 3
                                   '(gur history) '())
```

```
(make-entry 'bio101  'biology   '(squishy things) 3
                                '(gur biology) '())
(make-entry 'bio212  'biology   '(diseases and their applications) 4
                                '(gur biology) '())
(make-entry 'bio113  'biology   '(drugs and their applications) 4
                                '(gur biology) '())
(make-entry 'bio114  'biology   '(you and your brain) 4
                                '(gur biology) '())
(make-entry 'geo101  'geology   '(rocks for jocks) 4
                                '(rest) '())
(make-entry 'geo104  'geology   '(planets) 4
                                '() '(math203 phys102))
))
```

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

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

As explained in class, 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]

Here is 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 "$\langle s_1 \rangle$ $\langle s_2 \rangle \ldots$ and $\langle s_n \rangle$" where s_i are each in the catalog. See the attached code for how this restriction is implemented.

[4]Note, 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`.

```
(define subject-knowledge
  (list
   (make-rule
    `(what is (? s ,in-catalog) about)
    (lambda (dict)
      (let ((entry (value 's dict)))
        (write-line
         (append (list (entry-subject entry))
                 '(is about)
                 (entry-summary entry))))) )
   (make-rule
    `(what are (?? s ,subjects) about)
    (lambda (dict)
      (for-each (lambda (entry)
                  (write-line
                   (append (list (entry-subject entry))
                           '(is about)
                           (entry-summary entry))))
                (value 's dict))) )
   (make-rule
    `(how many credits is (? s ,in-catalog))
    (lambda (dict)
      (let ((entry (value 's dict)))
        (write-line
         (append (list (entry-subject entry))
                 '(is a)
                 (list (entry-credits entry))
                 '(credit subject))))) )
   (make-rule
    `(how many credits are (?? s ,subjects))
    (lambda (dict)
      (for-each (lambda (entry)
                  (write-line
                   (append (list (entry-subject entry))
                           '(is a)
                           (list (entry-credits entry))
                           '(credit subject))))
                (value 's dict))) )
   (make-rule
    `(what are the prerequisites for (?? s ,subjects))
    (lambda (dict)
      (for-each (lambda (entry)
                  (write-line
                   (append '(the prerequisites for)
                           (list (entry-subject entry))
                           '(are)
                           (entry-prerequisites entry))))
                (value 's dict))) )
   (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))))) )
   ))
```

2. Paper Exercises

You should write up the answers to the questions in this section before starting to write any code. Some preliminary practice with these ideas should prove extremely helpful in doing the code-writing exercises.

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.

list-union => (1 2 3 4)
append (1 2 3 3 4)

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

The result of list-union function doesn't include common elements of L1 and L2. e.g L1 (1 2 3) L2 (3 4)

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,

```
(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.

(reduce * 1 '(1 2 3 4 5 6 7 8 9 10))

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

(reduce + 0 (map (λ(x) (square x)) list))

Exercise 4: What result is returned by

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

(1 2 2 3 4 6 5 4)

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

(1 2 3 6 5 4)

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

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)))
```

Expel the repeated value of the list.

3. Code-Writing Exercises

Begin by loading the code for Problem Set 5. This will load the code from `ps5match.scm` and `ps5adv.scm`. You should copy the latter to your local directory using a command that looks like:

```
cp /home/sicp/psets/ps5/ps5adv.scm ~/usicp/work/
```

and use that copy to make your local edits. Start the advisor program by typing

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

and try some sample questions.[6]

Exercise 6: Expand the advisor's store of `beginnings` and `general-advice`. 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 prerequisites for GEO104 are given as MATH203 and PHYS102. But certainly to take GEO104 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 GEO104 to be `(math203 phys102 phys101 math102 math101)` (although not necessarily in that order). (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`.)

hard.

Exercise 9: Using `all-prerequisites`, add a new rule to `subject-knowledge` that answers questions of the form "Can I take ⟨*subject*$_1$⟩ if I have not taken ⟨*subject*$_2$⟩?" Turn in your rule and a sample showing that it works.

Exercise 10: Write a procedure `check-circular-prerequisites?` that takes a list of subjects and checks if any of the subjects are prerequisites (as given by `all-prerequisites`) for any other subject in the list. For example, you cannot take MATH101 and GEO104 simultaneously. Your procedure should return `false` if there are circular prerequisites, and `true` otherwise. Turn in your code and an example that shows your procedure works. (Hint: As in the hint for Exercise 8, try to do this with `list-union`, `list-intersection`, and `reduce`.)

[6] WARNING! Terminate your inputs by typing `C-x C-e`. Do **not** use `M-z`.

Exercise 11: 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 `(eecs101 math101 phys101)` the total is 13 credits. Turn in your code and an example that shows your procedure works.

Exercise 12: Write a procedure `check-subject-list` that takes a list of subjects and checks it for circular prerequisites and for exceeding the freshman credit limit, which is 18 credits. If the number of credits exceeds the credit limit the procedure should print an appropriate message, and similarly print an appropriate message if there are circular prerequisites. Otherwise, it should print a message of the form "you need the following prerequisites ..." together with the union of the immediate prerequisites of the subjects in the list. Turn in your code and an example that shows your procedure works.

Exercise 13: Add a new rule to `subject-knowledge` for which the pattern is

```
'(I want to take (?? s ,subjects))
```

where the action calls your procedure from Exercise 12 to print the response. Notice that the restriction `subjects` is defined so that `(value 's dict)` will return the list of catalog entries for the indicated subjects. Turn in your rule and some interactions showing that it works.

Exercise 14: Design and implement some other improvement that extends the advisor's capabilities.

For example, the program currently does not use the information about the University requirements that the subjects satisfy such as GUR (General University Requirements) or GAK (Gratifying Alternative Knowledge). There is not much use of the subject summary information either. 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 Burgundy University community who you think could be usefully replaced by simple programs: a dean or two, the Registrar, lecturers, ...

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.

Another Contest!

Prizes will be awarded for the cleverest programs and dialogues turned in for this problem set. For some inspiration see `http://www.fury.com/aoliza`.