MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Sample problem set

FReshman advisor

For this week's problem set, we had designed a truly enlightening investigation of the properties of multi-tape sorting algorithms on medium-sized databases. However, as you'll see from the attached letter, the 6.001 class has just received an emergency request for help, and this has forced us to make a last-minute change in plans. We hope this will not prove too much of an inconvenience.

Massachusetts Institute of Technology
Office of Undergraduate Academic Affairs

October 4, 1995

Profs. Hal Abelson and Gerry Sussman
Department of Electrical Engineering and Computer Science

Dear Hal and Gerry:

You've probably suspected for some time now that I'm getting sick and tired of trying to coax faculty into serving as freshman advisors. We've tried jawboning, sending cute memos through Institute mail, and offering free dinners, but your august colleagues, it seems, don't want to have anything to do with students. Even when they do volunteer to advise freshmen, the advice they give isn't that good anyway. Half of them still don't understand the freshman credit limit or the difference between a REST and a HASS-D.

As part of MIT's effort to reengineer our core processes, our education quality team has decided to get rid of all this freshman advising stuff and replace it with a a computer program. You've got smart students in your 6.001 class. I'm sure they could write something that will be at least as useful as our current crop of advisors. Do you think you could ask some of them to try? We really need this by next week in order to plan the Dean's office budget for next year.

Traviss T. D. Merit
Associate Dean for Undergraduate Academic Affairs

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.[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. 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 18:03 instead of 18.03).[2]

```
(see-advisor 'hal)
(HI HAL)
(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)
(MIT HAS A LOT OF INTERESTING DEPARTMENTS)
** (what is 12:004 about)
(12:004 IS ABOUT PLANETS)
** (can I take 12:004)
(THE PREREQUISITES FOR 12:004 ARE 18:03 8:02)
** (can I take 18:03 and 6:001)
(6:001 IS TOO MUCH WORK FOR FRESHMEN -- WAIT UNTIL NEXT YEAR)
** (what are 5:11 7:012 and 18:02 about)
(5:11 IS ABOUT SMELLY ORGANIC CRUD AND GOOP)
(7:012 IS ABOUT DISEASES AND THEIR APPLICATIONS)
(18:02 IS ABOUT MULTIVARIATE CALCULUS)
** (what are the prerequisites for 18:03)
(THE PREREQUISITES FOR 18:03 ARE 18:02)
** (how many units are 18:02)
(18:02 IS A 12 UNIT SUBJECT)
** (can I take 63 units 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 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 four rudimentary methods for generating

[1] This 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.

[2] 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.

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 "MIT 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 lecture on October 5. One method uses the pattern matcher alone. The other uses the matcher in conjunction with a database drawn from the MIT 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))

(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. 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

[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 `ctrl-X ctrl-E`. The value returned by `prompt-for-command-expression` is the expression you type.

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
  '((make sure to take some humanities)
    (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))))
```

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 lecture. 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)`. [4]

[4] 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 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) 6:001 (?? y))
              (simple-response '(6:001 is too much work for freshmen
                                  -- wait until next year)))
   (make-rule '((?? x) 8:01 (?? y))
              (simple-response '(students really enjoy 8:01)))
   (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 lecture, 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 12:004 about)
(12:004 IS ABOUT PLANETS)
** (can I take 12:004)
(THE PREREQUISITES FOR 12:004 ARE 18:03 8:02)
```

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

```
(define catalog
  (list
   (make-entry '8:01 'physics '(classical mechanics) 12 '(GIR physics) '())
   (make-entry '8:02 'physics '(electricity and magnetism) 12 '(GIR physics) '(8:01 18:01))
   (make-entry '8:03 'physics '(waves) 12 '(rest) '(8:02 18:02))
   (make-entry '8:04 'physics '(quantum weirdness) 12 '(REST) '(8:03 18:03))
   (make-entry '18:01 'math '(elementary differential and integral calculus) 12 '(GIR calculus)
'())
   (make-entry '18:02 'math '(multivariate calculus) 12 '(GIR calculus) '(18:01))
   (make-entry '18:03 'math '(differential equations) 12 '(REST) '(18:02))
   (make-entry '18:04 'math '(theory of functions of a complex variable) 12 '() '(18:03))
   (make-entry '6:001 'eecs '(scheming with abelson and sussman) 15 '(REST) '(true-grit))
   (make-entry '6:002 'eecs '(circuits) 15 '(REST) '(8:02 18:02))
   (make-entry '3:091 'mms '(munching and crunching stuff) 12 '(GIR chemistry) '())
   (make-entry '5:11 'chemistry '(smelly organic crud and goop) 12 '(GIR chemistry)
'(a-strong-stomach))
   (make-entry '7:012 'biology '(diseases and their applications) 12 '(GIR biology) '())
   (make-entry '7:013 'biology '(diseases and their applications) 12 '(GIR biology) '())
   (make-entry '7:014 'biology '(diseases and their applications) 12 '(GIR biology) '())
   (make-entry '12:001 'eaps '(rocks for jocks) 12 '(REST) '())
   (make-entry '12:004 'eaps '(planets) 12 '() '(18:03 8:02)) ))
```

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))))
```

As explained in lecture, 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.[5]

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

[5]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 units is (? s ,in-catalog))
    (lambda (dict)
      (let ((entry (value 's dict)))
        (write-line
         (append (list (entry-subject entry))
                 '(is a)
                 (list (entry-units entry))
                 '(unit subject))))) )
   (make-rule
    `(how many units are (?? s ,subjects))
    (lambda (dict)
      (for-each (lambda (entry)
                  (write-line
                   (append (list (entry-subject entry))
                           '(is a)
                           (list (entry-units entry))
                           '(unit 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. To do before Lab

You should write up the answers to the questions in this section before coming to lab. Some preliminary practice with these ideas should prove extremely helpful in doing the lab 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.[6] `List-intersection`, similarly returns the intersection.

Pre-lab 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.

`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.

Pre-lab Exercise 2: Show how to use `reduce` to compute the product of all the elements in a list of numbers.

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

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

Pre-lab 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`?

Pre-lab 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. To do in Lab

Begin by loading the code for problem set 5. This will load the code from `match.scm` and `adv.scm`. It will also create a buffer for `adv.scm` so you can edit pieces of it.

Start the advisor program by typing

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

and try some sample questions.[7]

Lab Exercise 1: Expand the advisor's store of `beginnings` and `general-advice`. You may find it helpful to consult MIT course evaluation guides—or fabricate your own helpful advice. Turn in a short demonstration transcript of the conversation.

Lab Exercise 2: 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.

Lab Exercise 3: The `entry-prerequisites` procedure used by the advisor returns the list of prerequisite subjects as listed in the catalog. For example, the prerequisites for 12:004 are given as 18:03 and 8:02. But certainly to take 12:004 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

[7] WARNING! Terminate your inputs by typing `ctl-X ctl-E`. Do **not** use `M-z`.

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 12:004 to be `(18:03 8:02 8:01 18:02 18:01)` (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`.)

Lab Exercise 4: 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.

Lab Exercise 5: 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 18:01 and 12:004 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 3, try to do this with `list-union`, `list-intersection`, and `reduce`.)

Lab Exercise 6: Write a procedure `total-units` that takes a list of subjects and returns the total number of units of all the subjects in the list. For example, if the list is `(6:001 18:01 8:01)` the total is 39 units. Turn in your code and an example that shows your procedure works.

Lab Exercise 7: 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 54 units. If the number of units 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.

Lab Exercise 8: 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 7 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.

Lab Exercise 9: Design and implement some other improvement that extends the advisor's capabilities.

For example, the program currently does not use the information about the Institute requirements that the subjects satisfy such as GIR or HASS. 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 MIT community who you think could be usefully replaced by simple programs: a dean or two, the Registrar, your 6.001 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.