MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1989

Problem set 3

Issued:	Tuesday 26 September
Due:	Wednesday 4 October
Reading:	Text, Chapter 2, sections 2.1 and 2.2.

Problem sets should always be handed in at recitation. Late work will not be accepted.

Homework exercises

Write up and turn in the following exercises from the text:

- Exercise 2.3, page 84.
- Exercise 2.6, page 86.
- Exercise 2.7, page 86.
- Exercise 2.11, page 87.

Programming Assignment: Star Gazing

One of Louis Reasoner's frequent habits is taking walks in the countryside late at night – to help him "clear his head," he claims. Unfortunately, Louis often loses his orientation, and has some difficulty in returning to his car. He decides he needs help in orienting himself, and plans to invent a device to help him use the stars as a guide.

Suppose that Louis wants to know what part of the sky he is seeing. This can be accomplished by recognizing some familiar pattern of stars. There are two parts to the process of using the stars for orientation:

- Recognizing spots of light in the sky as known groups of stars. Here we match the bright spots in an image of the sky against star charts representing traditional configurations, called constellations.
- Determining how a recognized group is oriented with respect to the matching chart. Such a relation is called a transformation of coordinates.

These two tasks are closely interrelated: it is hard to do one without the other – in other words, if we have solved either of the two problems we have solved the other.

In this problem set, we are going to help Louis develop a method for recognizing constellations, given charts of those constellations and images of the stars. The methods developed here actually

have less "blue sky" applications: We can think of the "spots" as feature points in an image of something other than the sky. These feature points are places in the image that are especially easy to identify, such as the corners of polygonal regions of constant brightness. The images might be of a flat industrial part on a conveyor belt, for example, taken with a camera that is oriented with its optical axis perpendicular to the belt. In fact, the methods we are going to explore in this problem set are a simple version of a general purpose object recognition system called RAF, developed by Eric Grimson and Tomás Lozano-Pérez.

Louis begins by trying to understand his data. He calls the points of light in an image of the sky "spots" to distinguish them from the "stars" on a chart. Each star on a chart has a name, and Louis chooses to give each spot on an image an identifying number.

To simplify the algebra, Louis pretends that the sky is flat. He describes the position of each spot in the sky and each representation of a star on a chart using two coordinates.

In addition, every point of light in an image and every star on a chart has a brightness, measured in magnitudes (to be explained later).

For example, a simple star chart might contain the following entries:

Name	*X*	*Y*	*Magnitude*
alpha	0.0	1.0	0.0
beta	0.0	0.0	0.5
gamma	1.5	1.0	1.0
delta	1.0	2.0	2.0

and a simple image might look as follows:

Number	*X*	*Y*	*Brightness*
1	1.03	1.97	0.2
2	1.32	3.75	1.2
3	0.25	2.75	0.3

Louis begins by defining constructors for these abstractions:

```
(define make-star
  (lambda (star-name star-position star-magnitude)
    (list star-name star-position star-magnitude)))

(define make-spot
  (lambda (spot-number spot-position spot-brightness)
    (list spot-number spot-position spot-brightness)))

(define make-position
   (lambda (x y) (list x y)))
```

Exercise 1: Louis represents a star chart as a list of stars. Louis makes a simple test chart, called `trial-stars-1`, follows:

```
(define trial-stars-1
  (list (make-star 'alpha (make-position 0.0 1.0) 0.0)
        (make-star 'beta (make-position 0.0 0.0) 0.5)
        (make-star 'gamma (make-position 1.5 1.0) 1.0)))
```

Draw a box-and-pointer diagram for the list structure corresponding to `trial-stars-1`.

Exercise 2 Given these constructors, we need to define appropriate selectors for each of the components of the compound data items. Complete the definitions for the selectors below:

```
;;; Given a star, we should be able to get its components

(define star-name ...)                          ;should get star's name

(define star-position ...)                      ;should get star's position vector

(define star-magnitude ...)                     ;should get star's magnitude


;;; Given a spot, we should be able to get its components

(define spot-number ...)                        ;should get spot's number

(define spot-position ...)                      ;should get spot's position vector

(define spot-brightness ...)                    ;should get spot's v coord
```

Louis now attacks the problem of matching a list of spots to a list of stars. Louis calls a particular matching, mapping every spot observed to a star (from his chart), an *interpretation*. Louis's idea is that he can first develop all possible assignments of spots to stars, and later filter them for reasonableness. His first draft of a matcher is thus nothing more than a program that makes (a list of) all possible interpretations of an image (a list of spots) relative to a chart (a list of stars), without taking into account any of the constraints implied by positions and brightnesses. Below we find Louis's first draft:

```
(define (all-interpretations image chart)
  (if (null? image)
      (list the-empty-interpretation)   ;Note1
      (let ((spot-to-match (first image))
            (spots (rest image)))
        (define (match-all stars)
          (if (null? stars)
              the-empty-list            ;Note2
              (append (extend-interpretations
                        (pair-up spot-to-match (car stars))
                        (all-interpretations spots
                                             (delete (car stars) chart)))
                      (match-all (cdr stars)))))
        (match-all chart))))

(define (extend-interpretations pair interpretations)
  (define (extend-all interps)
    (if (null? interps)
        the-empty-list
        (cons (augment-interpretation pair (car interps))
              (extend-all (cdr interps)))))
  (extend-all interpretations))

;;; My data abstraction implementations are below this line.

(define the-empty-list '())     ;Ben B. says this is good magic.
(define first car)
(define rest cdr)

(define the-empty-interpretation the-empty-list)
```

```
(define augment-interpretation cons)

(define pair-up list)
(define pair-spot car)
(define pair-star cadr)
```

The set of interpretations of a set of spots relative to a set of stars is formed as follows. The first spot can be assigned to any star. This can be the beginning of any interpretation that matches the rest of the spots to the stars not yet used.

Louis has marked a few subtle points: Note1 reminds us that there is one interpretation of no spots, the empty interpretation! Note2 reminds us that if there are spots to be matched but no stars left on the chart there are no consistent interpretations.

When you load problem set 3 into your Chipmunk, copies of these procedures are also loaded into your editor, for your convenience, as are some utility procedures for printing out interpretations, `print-match` and `print-matches`. In addition, an example list of stars, called `trial-stars-1`, and an example list of image spots, called `trial-spots-1`, are also loaded. You may also find the procedure `print-matches` useful for printing out the results of finding the matches in a readable form.

You can try it out:

```
(all-interpretations trial-spots-1 trial-stars-1)
```

You will see that Louis's procedures will find all possible ways of matching stars with spots, without any constraint on the matches. So if there are m stars and n spots (with $m \geq n$), we will generate $\frac{m!}{(m-n)!}$ interpretations. These are all the ways of picking spots to pair up with stars. When $m = n$, this procedure provides us with a convenient way of generating all $m!$ permutations of m elements.

Louis needs to pick out the right matching. He could do this by checking all of the possible interpretations for the best one, but because factorials grow very fast he needs some way of reducing the number of interpretations that will be generated.

One way that Louis can reduce the number of possible interpretations is to take brightness into account when matching stars and spots. Stars have different magnitudes (note that in astronomy, the magnitude of a star is a measure of its brightness, and is measured on a logarithmic scale, where a difference of five magnitudes corresponds to a ratio of 100 in brightness). Correspondingly, spots in the image will have different brightnesses. Measurements of brightness cannot be made with perfect accuracy, so we have to allow for the possibility of a small difference when we test whether a spot could match a star.

Exercise 3: Write a procedure called `similar-magnitudes?`, which takes as arguments a star and a spot, and which evaluates to true if the absolute difference between the magnitude of the star and the brightness of the spot is less than the value of a global variable, `magnitude-tolerance`, (which we have defined initially to be 1.0 in the code file). Be sure to use appropriate abstractions and their relevant selectors and/or constructors.

To use this information requires modification of the first-draft interpretation generator. The current code assumes that any pairing is acceptable.

Exercise 4:

You are to change Louis's code to require that for a spot to be paired with a star, they must have similar magnitudes. The change required is very simple, only requiring the addition of a few lines of code to the procedure `all-interpretations`.

Try out your new matcher using the lists of stars and spots you created in Exercise 1. You should notice that adding the constraint of similar magnitude can decrease the number of matches explored, provided that the error in brightness measurements is small enough. In fact, if brightness could be measured with unlimited accuracy, Louis's job would be done, since a star's magnitude would then be a unique identifying characteristic. In practice, however, measurements of brightness have limited accuracy and the procedure may find several potential matches that each pass the brightness comparison tests.

Hand in your modified code and demonstrate, using appropriate examples, that it does "the right thing".

We need to further prune the number of possible interpretations, and we can do so by considering information about the spatial relationship between the stars in a constellation. So far, we have only used a unary test, one that applies to a single star and a single spot. Now let use consider a binary test. The distance between stars is not changed by rotation or translation in the image plane. So distance is independent of the coordinate transformation we are seeking, and thus comparison of distances between stars and corresponding spots gives us a convenient additional filtering test for proposed partial matches. Note that this binary test is more expensive than the unary brightness test, since we have to compare the distances between the proposed new star and all of the stars already included in the partial match with the corresponding distances between the proposed new spot and all of the spots already included in the partial match. If these distances can be measured with high accuracy, however, then this test will remove almost all erroneous partial matches.

Exercise 5: We need procedures for computing distances between spots and between stars. Write a procedure, `distance`, that computes the distance between two two-dimensional points on a plane. Also write a procedure, `similar-distances`, which takes as arguments two pairs, each of which is a pairing of a spot and a star, and which tests whether the absolute difference in distances between the respective stars and spots is less than a globally specified `distance-tolerance` defined as follows:

```
(define distance-tolerance 0.5)
```

Exercise 6: We now want to use the distances between pairs of stars and the corresponding pairs of spots to prevent the generation of impossible matches. Given a partial (or current) interpretation and a new pairing of a star and a spot, we want to ensure that the distance between the new star and each of the stars in the interpretation is consistent with the distance between the new spot and each of the spots in the interpretation. Write a procedure called `distances-check`, that takes a new pair (consisting of a new star and a new spot) and a current interpretation, (consisting of a list of such pairs), and returns true if the new pair has similar distances with all the pairs in the interpretation, and false otherwise.

Exercise 7: Using these procedures, we need to modify Louis's code to incorporate this distance constraint. The change required is very simple, only requiring the addition of a few lines of code to the procedure `extend-interpretations`. You may also find it instructive to see how the number of matches changes with the amount of error allowed in the match. Try varying the global parameter `distance-tolerance`, which initially has the value 0.5, making this value smaller until only one match of stars to spots is possible. What is the actual match in this case?

Figure 1: To transform a set of points in image coordinates into model coordinates, we first rotate the points by an angle θ, and then translate the result.

If the accuracy of image position measurement is high, the procedures we have built will typically find the correct match, and only the correct match, unless the constellation has an unusually symmetric spatial pattern. If, on the other hand, the accuracy is limited, there may be several different matches between stars and spots that pass the pairwise distance test. Also, some of the matches that pass the pairwise distance test may not make sense globally. Consider, for example, a match between a constellation and some spots arranged in a pattern that is a mirror-image of the constellation. There is in general no rotation and translation that will bring the constellation and the set of spots into alignment, yet all of the pairwise distances will match.

We use u and v as coordinates in an image, while x and y are coordinates in some global coordinate system in which all of the constellation models are described. In our flat sky model, the transformation from one to the other involves a rotation and a translation:

$$\begin{aligned} x &= c\,u - s\,v + x_0 \\ y &= s\,u + c\,v + y_0 \end{aligned}$$

where $c = \cos\theta$ and $s = \sin\theta$ where θ is the angle between corresponding axes in the two coordinate systems. Further, x_0 and y_0 are the offsets of the origin of the image coordinate system in the model coordinate system. This transformation is illustrated in Figure 1. Part of our matching task is the recovery of the transformation parameters, c, s, x_0 and y_0.

This means that when we have a possible matching of all the stars to spots, we should estimate the transformation from the image to the model and then verify that this transformation maps all of the spots to positions near the stars that they have been matched with. A transformation returns c, s, x_0 and y_0.

We have provided a procedure called `compute-transformation`, that will compute such a transformation, given as arguments `xa ya xb yb ua va ub vb`, where `xa, ya` are the x and y coordinates of one star, `ua, va` are the u and v coordinates of the matching spot, and `xb, yb, ub, vb` are the coordinates for a second matching star and spot. `Compute-transformation` returns a list of `c, s, x0, y0` corresponding to the components of the computed transform.

Further we have provided a procedure called `map-uv-to-xy` that, given as arguments `u` and `v` as the coordinates of a point in an image, plus a transform (as returned by `compute-transformation`),

returns a pair (the result of using a `cons`) giving the coordinates for the transformed point in the sky's coordinate system, obtained by transforming the image point by the transform.

Exercise 8: To guarantee that the matching of spots to stars is correct, we need to use the computed transform to test that each image spot, when transformed by the transformation, is close to its matching star. Write a procedure called `check-matched-pairs`, which takes as arguments an interpretation (i.e. a list of matches of star and spot) and a transformation, and returns true if for all of the matches, the transformed image spot is within `distance-tolerance` of its corresponding model star.

Exercise 9: Change your matcher to include the procedures `compute-transformation` and `check-matched-pairs` to filter the list of proposed interpretations so that any that are returned are in fact globally correct.