Programming Assignment: Navigating Through Boston

Since arriving in Boston, Louis Reasoner has been frustrated by his inability to navigate the streets of Boston with any efficiency or consistency. “Too many one-way streets,” he grumbles, “and besides, they never put street signs on the big streets, only on the teeny-tiny ones that dead-end after a block.” To help find out where he is when he gets lost, Louis decides to build an automatic navigator, but as usual, Louis needs a little help from his friends, especially you.

Louis buys an atlas of the city, represented by street segments. Each segment is described by two endpoints, measured in an appropriate unit (e.g. Smoots) in a coordinate frame centered at the Great Dome and oriented to true north. A segment consists of the portion of a street between consecutive intersections.

Louis then outfits his car with an automated odometer which measures the distance traveled from one intersection to another in Smoots, and a gyroscope which measures the angle turned at an intersection. Hence Louis’ onboard computer can report the \((x, y)\) position of each intersection he reaches. Unfortunately, the coordinate system in which he measures his trip is different from the one in his atlas; it has unknown origin and rotation, since he is lost when he starts the trip.

Thus, Louis is confronted with the following situation. He has an atlas, composed of a set of street segments, each labeled by a street name, and the positions of the start point and end point of the segment, measured in one coordinate frame. He also has information about the actual street segments he has travelled, which might include a name (though not always) and which include the positions of the start point and end point of the segments, but measured in a different coordinate frame.

Louis’ plan is to match properties of the street segments he travels with the corresponding properties of segments from the atlas, in order to deduce exactly where he is in the city. The idea is that if he travels far enough, there should only be one way to match his trajectory of segments against the atlas, and this will let him figure out where he is.

Data Abstractions

Louis begins by trying to make sense of his data. He calls the street segments on his atlas atlas-segments to distinguish them from the trip-segments that he actually travels. Each segment has a (perhaps non-unique) name, as well as two endpoints, and each point has an \(x\) and \(y\) coordinate.
Louis begins by defining constructors for these abstractions:

```
(define make-segment ; string, point, point -> segment
  (lambda (name start finish) (list name start finish)))

(define make-point ; number, number -> point
  (lambda (x y) (list x y)))
```

**Exercise 1:** Louis represents an atlas and a trip as lists of segments:

```
(define make-atlas list) ; segment, ... -> list<segment>
(define make-trip list) ; segment, ... -> list<segment>
```

Note that the segments in a trip and an atlas appear in an arbitrary order. In particular, the segments in a trip do **not** appear in the order in which Louis drove through them.

**Note that these definitions are provided for you in the file routes_code.scm which you can download from the web site and save.**

Louis makes a simple test atlas, called `trial-atlas-0`, as follows:

```
(define trial-atlas-segment-1
  (make-segment "beacon" (make-point 0 0) (make-point 0 2)))

(define trial-atlas-segment-2
  (make-segment "cambridge" (make-point 0 2) (make-point 3 5)))

(define trial-atlas-0
  (make-atlas trial-atlas-segment-1 trial-atlas-segment-2))
```

Draw a box-and-pointer diagram for the list structure corresponding to `trial-atlas-0`. (You don’t actually have to turn this part in, but we strongly encourage you to do this problem, so you have a clear understanding of the data structures you are using.)

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

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

(define segment-name ...) ; segment -> string ; get name of a given segment
(define segment-start-point ...) ; segment -> point
```
In the definitions above, `segment-start-point` returns the starting point of a given segment. One could also build a selector that takes a `segment` as input, and returns the x coordinate of the starting point. Implement `segment-start-x` and `segment-end-x` (without violating data abstraction). Similarly, implement `segment-start-y` and `segment-end-y`. Turn in a transcript showing your testing of these constructors and selectors. Also, be sure to document your code appropriately!

**Matching algorithm for relating trips and altases**

In the rest of this project, you will write progressively better algorithms to help Louis match trip-segments that he has traveled to his atlas. As is common in software development, you will first solve a simpler problem, then iteratively improve on your solution.

First, a little terminology. We will call a pairing of a trip-segment and an atlas-segment, a *match*. We will call a set of matches, one for each segment in the trip, a *correspondence*.

One way to proceed would be to find all possible correspondences, and then remove those that don’t make sense. The problem is that if the atlas is big, this becomes a huge set. For example, if there are $m$ atlas-segments and $n$ trip-segments (with $m \geq n$), then there are $\frac{m!}{(m-n)!}$ correspondences. If $m$ is large, this is roughly $m^n$, which can be huge.

So an alternative to generating all possible correspondences is to sample the set of possible correspondences, and rely on statistics to help us find solutions. The idea is to randomly sample possible correspondences and test them. If only one correspondence is consistent, then it is probably correct (probably, because we haven’t necessarily tested all correspondences). If we try this for larger and larger samples of the space of correspondences, and still only get one solution, the probability that we know our location increases.

Here is a simple way to make a correspondence. First, we have a constructor for making matches, called `make-match` (see the code). Using this, we could build an explicit data abstraction for correspondences, but for now we will just rely on using lists:

```scheme
(define (make-correspondence trip atlas)
  (cond ((null? trip) nil)
        (else (cons (make-match (car trip) (car atlas))
                    (make-correspondence (cdr trip) (cdr atlas))))))
```

Of course, given a trip and an atlas, this will always create the same correspondence. But we can use this as a basis for a more useful method:
(define (make-correspondence trip atlas)
  (cond ((null? trip) nil)
        (else (let ((other (select atlas)))
               (cons (make-match (car trip) other)
                     (make-correspondence (cdr trip)
                                          (remove-element other atlas)))))))

The idea in this version is to select a segment from the atlas at random, and match it to the next segment of the trip. This match is added to whatever we get by finding a correspondence between the remainder of the trip and the atlas without that matched segment. (Note that we are assuming that Louis never backtracks as he drives around Boston. This is not completely realistic, but it simplifies our project.)

We have provided an abstraction for matches in the code, with constructor `make-match` and selectors `first-of-match`, `second-of-match`.

**Exercise 3:** Complete the implementation for generating correspondences by providing the `select` procedure and the `remove-element` procedure. The `select` procedure should take a list as input, and return one of the elements of that list, chosen at random. Remember that `random` is a Scheme procedure which, given a positive integer argument `n`, returns an integer between 0 and `n-1` chosen at random.

The `remove-element` procedure should take an element and list as input, and return a new list, which contains all the elements of the original list, except the chosen element. Note that since we know the elements we are considering are lists, we can use `eq?` or `equal?` to test equality. Write the procedure `remove-element` using `filter` (and without using `delq`).

Show your procedure `make-correspondence` running on a set of test cases, using `trial-trip-1` and `trial-atlas-1`. These are examples of a trip and atlas that are defined when you load `routes_code.scm` into your Scheme environment.

Note how your procedure, `make-correspondence`, (assuming you wrote it correctly) generates different correspondences for the trip.

**Exercise 4:** Once we have the idea of generating a correspondence, we can use it to create samples from the space of all possible correspondences. Write a procedure, called `generate-correspondence-set` with calling form:

`(generate-correspondence-set trip atlas set n)`

This procedure takes as argument: a trip, an atlas, a set of correspondences represented as a list (initially empty), and a positive integer, indicating the number of trials to make. The basic idea is to call `make-correspondence` `n` times, and collect the results into a list, which is returned. The nuance is that if a correspondence is already in the set, we don’t include it. Thus, if we run our procedure with 100 trials, we might get fewer than 100 correspondences in the set, but the correspondences we get in our set are distinct.

Write this procedure, and any other procedures you need to support it. Note that in testing whether two correspondences are the same, you can take advantage of the fact that the trip-segments are
in the same order in each correspondence, so that you only have to test equality of the atlas part of each match in a correspondence.

To test equality of correspondences, you have a choice. You could take advantage of the fact that we are using lists to represent our structures, and use \texttt{equal?} to test equality of list structures. Alternatively, you could create data abstractions for matches and correspondences, and create predicates to go with your data abstractions.

Show your code running on:

\begin{verbatim}
(define foo (generate-correspondence-set trial-trip-1 trial-atlas-1 nil 10))
(define foo (generate-correspondence-set trial-trip-1 trial-atlas-1 nil 100))
\end{verbatim}

How many correspondences do you get in each case, and what is the maximum possible?

Do the same with \texttt{trial-trip-2} and samples of 10, 100, 500. How many correspondences do you get, and what is the maximum possible?

\textbf{Filtered matching}

You saw that \texttt{generate-correspondence-set} can produce a large output. This just gives a set of possible correspondences, but many of them don’t make sense. We need to remove correspondences that don’t make sense, and we would like to intertwine this testing of correspondences with generation (i.e. we don’t want to make a huge set then filter it). What does it mean to “make sense”? For example, Louis can take the street name into account when matching trip-segments to atlas-segments.

This approach isn’t foolproof. Many atlas-segments have the same name, because every block of each street appears in the atlas as a different segment. Also, Louis doesn’t always see the name of a street when he drives along it; an unnamed trip-segment matches all atlas-segments. Still, name matching can substantially reduce the number of correspondences generated.

To implement name matching, the first thing to do is to extend the \texttt{segment} data abstraction to support segments with unknown names. Extending the data abstraction requires adding a new constructor and predicate.

\begin{verbatim}
(define make-unnamed-segment ; point, point -> segment
  (lambda (start finish) (list internal-name-for-unnamed-segment start finish)))

(define unnamed-segment? ; segment -> boolean
  (lambda (segment) (string=? (car segment) internal-name-for-unnamed-segment)))

(define internal-name-for-unnamed-segment "###")
\end{verbatim}

The variable \texttt{internal-name-for-unnamed-segment} is internal to the data abstraction and should not be used externally. Later in the course we will show you a way to implement the data abstraction that prevents private data like \texttt{internal-name-for-unnamed-segment} from being used externally; for now, it is internal only by convention.
Exercise 5: Write a procedure called \texttt{same-name?}, which takes as argument a \texttt{match} containing two \texttt{segments} and which evaluates to true if the names of the two \texttt{segments} are the same, or if one of the segments is unnamed. Be sure to use appropriate abstractions and their relevant selectors and/or constructors.

\begin{verbatim}
; \texttt{same-name?}: evaluates to true if the two segments have the same name or one is unnamed
; type: \texttt{match<segment,segment> \rightarrow boolean}
(define \texttt{same-name?} ...) 
\end{verbatim}

Exercise 6: Write a procedure \texttt{always?}, which takes a predicate (that is, a function that returns a boolean value) and a list of elements as input. \texttt{Always?} returns \texttt{true} if applying the \texttt{predicate} to each element in the list returns \texttt{true}, otherwise it returns \texttt{false}.

\begin{verbatim}
; \texttt{always?}: evaluates to true if a correspondence satisfies a predicate for each match
; type: ((\texttt{match<segment,segment> \rightarrow boolean}), \texttt{list<match<segment,segment>}}) \rightarrow boolean
(define \texttt{always?} ...) 
\end{verbatim}

; for example
(always? \texttt{integer?} (list 1 2 3 4))
;Value: \texttt{#t}
(always? \texttt{odd?} (list 1 2 3 4))
;Value: \texttt{#f}

Exercise 7: Now, create a procedure \texttt{generate-constrained-correspondence-set}. This should look very much like \texttt{generate-correspondence-set}, the difference is that it also takes a \texttt{predicate} as argument, and applies that \texttt{predicate} to each new correspondence to see if it should be kept. If the correspondence passes the predicate test, and is not already in the set, then it is added to the set.

Try the same tests you did in Exercise 4, but now using the constrained version of the generator, and \texttt{same-name?} as the test. Note that since \texttt{same-name?} applies to matches, and we want our predicate to apply to correspondences, you will want to use \texttt{always?} as part of your predicate to apply \texttt{same-name?} to each element of the correspondence. How many possible correspondences do you find in this case for 10, 100, 500, 1000 samples?

Exercise 8: Further filter the set of possible correspondences by also rejecting proposed matches of trip-segments and atlas-segments if the lengths of the two segments are different.

To do this, write a new predicate \texttt{same-name-and-length?} that, in addition to checking for \texttt{same-name?}, also compares the lengths of the trip-segment and atlas-segment. Thus, it’s type would be \texttt{match<segment,segment> \rightarrow boolean}. There is no need to take the square root to compute the actual lengths; it is sufficient to compare the squares of the lengths.

Make sure to use appropriate selectors when accessing the endpoints and their coordinates.
Try out your new predicate by rerunning the experiments from Exercise 7, with both `trial-trip-1` and `trial-trip-2`. You should notice that adding the length constraint further decreases the number of matches explored. Extend the testing to include 5000 samples, 10000 samples. How many correspondences do you get?

### Filtering over wider scope

So far, we have only used unary predicates, ones that apply to a single proposed match. Now let us consider a predicate that covers the entire correspondence under consideration.

Suppose we consider any two trip-segments in Louis’ trip. The direction of one trip-segment relative to the other must be the same as the corresponding relative direction of matching atlas-segments. For example, if a given trip-segment A is oriented at 90 degrees to another trip-segment B, then the atlas-segment matching A must also be at 90 degrees from the atlas-segment matching B. We want to incorporate such a test into our system.

**Exercise 9:** We need procedures for computing angles between atlas-segments and between trip-segments. Write a procedure, `angle-segments`, that takes as arguments two segments and returns the angle between them.

To implement this function, first compute a vector for each segment by subtracting the coordinates of the start point from the coordinates of the end point of the segment. Note that you can represent this vector as a point, using your earlier constructor. Given a vector (represented as a point) for each segment, compute the angle between them with the following formula

$$\text{rotation from } v_1 \text{ to } v_2 = \arctan \frac{v_1 \times v_2}{v_1 \cdot v_2}$$

If \( v_1 = (x_1, y_1) \) and \( v_2 = (x_2, y_2) \) are two 2D vectors, then the cross product \( v_1 \times v_2 \) is given by

\[
(x_1y_2) - (x_2y_1)
\]

and the dot product \( u \cdot v \) is given by

\[
(x_1x_2) + (y_1y_2)
\]

Scheme has a builtin function `atan` that computes the arctangent.

Note that the dot product is 0 if the argument vectors are orthogonal. Your implementation should handle this case without error.

Show tests that demonstrate that `angle-segments` is implemented correctly.

Finally, write a procedure `similar-angles?` that returns true if the relative angle between two trip-segments is the same as the relative angle between two atlas-segments, given two matches each containing a trip-segment and an atlas-segment.

```scheme
; similar-angles?: returns true if the relative angle between the
; first segments of the matches is the same as the relative angle
; between the second segments of the matches.
;
; type: match<segment,segment>, match<segment,segment> -> boolean
(define similar-angles? (lambda (match1 match2) ...))
```
Exercise 10: We now want to use the angle between pairs of trip-segments and the corresponding pairs of atlas-segments to help prevent the generation of impossible matches. Given a correspondence, we want to ensure that the relative angle between pairs of trip-segments in the correspondence is consistent with the angle between pairs of atlas-segments in the correspondence. Write a predicate that returns true if all the relative angles are similar, or false if any of them don’t match. Note that as opposed to earlier constraints that applied to single matches of trip-segments and atlas-segments, this applies to pairs of matches. Hence, your predicate will need to apply to a full correspondence, and measure relative angles of pairs of matches.

Exercise 11: Try out your new predicate by rerunning the experiments from Exercise 7, with both trial-trip-1 and trial-trip-2. You should notice that adding the angle constraint further decreases the number of matches explored. Extend the testing to include 5000 samples, 10000 samples. How many correspondences do you get?

If you are daring, try running this with a much larger size sample, say 100000.

Exercise 12 (open ended): Now that you have explored locating yourself using travelled segment information, try extending the system. In particular, create a set of procedures with the following behavior:

- Your main procedure will be something that generates a trip at random. It should take as input a current point (which you should select from one of the segments that makes up the atlas), a trip (which is the set of segments you have travelled so for – initially empty), and an atlas.
- It should find the set of possible segments you could travel from this point, by finding those segments in the atlas that have the current point as one of the endpoints, but which are not in the trip (we are going to avoid loops).
- If there are no possible segments, it should return the trip, which you can then run through your code to see if you can figure out where you are.
- Otherwise it should pick one at random, and continue until either a sufficient number of segments has been added, or there are no other choices (you get to decide what sufficient means).

Demonstrate your procedure on some trial cases, and show that the trips it generates can be matched against an atlas.

Then, try some variations:

- when you go to add a segment to your trip, randomly decide whether the street sign is visible or not. If not, add an unnamed segment with the right endpoints to the trip, else add the chosen segment.
- change the trip generation system so that you get to provide input on which choice to take at each step (which means you will need some way of displaying the options, and some way of reading in your choice).
- change the system to allow backtracking along segments, and see how it varies.
- try something else.
Finally

Please indicate the names of any students with whom you collaborated in doing this assignment. Please check the course policy on collaboration to be sure that you are within its constraints.