

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 2005

Project 1

Issued: Monday, September 19th
Due: Friday, September 30th, by 6:00PM
Code: `boston_travel.scm`

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. Mega-Smoots) in a coordinate frame centered at the Great Dome and oriented to true north. In other words, an endpoint is a combination of an x and y coordinate, measured in some units, from some arbitrary origin. 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 Mega-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.

One way to think about this is as a graph matching problem. That is, Louis has two sets of line segments, one representing an atlas or map of the city, and one representing the roads he has traveled. His goal is to figure out how to line up the two sets of lines so that they match – if he can do that, he can then 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.

You will hear a lot about data abstractions, and constructors and selectors in lecture on Thursday, September 22nd. If you want to get started on the project before then, you might either read the appropriate section of the textbook, or view the online lectures and lecture notes, or read the following brief description.

A data abstraction is just a way of gluing things together into a new thing that you can treat as an atomic unit. Associated with a data abstraction is a **constructor** – a way of taking two or more parts and gluing them together; and some **selectors** – procedures that take one of these abstractions and pull apart a desired piece. The most common data abstraction in Scheme is a **list**. It has a constructor, called `list`, which takes an arbitrary number of arguments, and creates an ordered sequence of those arguments. For example:

```
(define my-test (list 1 2 3 4 5))
```

To get the parts back out of a list, we can use a built-in selector, called `list-ref`, for example:

```
>(list-ref my-test 0)
1
```

```
>(list-ref my-test 3)
4
```

Note that what you might think of as the first element of the list is referenced as the zeroth element.

Now, Louis begins by defining constructors for his 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)))
```

The notation in the comment is that a segment consists of three parts, a name (which will be of type `string`, and two points; a point consists of two parts, an x and a y coordinate, each of which is a number. We haven't seen strings before, these are just sequences of characters, enclosed in double quotes, such as

```
"this is a string"
```

Strings come with their own sets of procedures for comparison, for example, to compare two strings, one uses


```
;; given a Point, we need its components

(define x-coor ...) ; Point -> number

(define y-coor ...) ; Point -> number
```

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`. Include a transcript showing your testing of these constructors and selectors. Also, be sure to document your code appropriately!

Matching algorithm for relating trips and atlases

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 can build an explicit data abstraction for correspondences:

- To add a leg to a correspondence, we will need a procedure called `add-leg`;
- To get the first leg of a correspondence, we will need a selector called `first-leg`;
- To get everything but the first leg of a correspondence, we will need a selector called `rest-correspondence`;
- We will need something that represents an empty correspondence, or one with no elements, called `empty-correspondence`;
- And we will need a way of testing if a correspondence is empty, called `empty-correspondence?`

Exercise 3: Create the necessary elements of a correspondence, as described above.

Now one way we could create a correspondence, from a trip and an atlas would be the following, where we rely on the fact that a trip and an atlas are each represented as lists.

```
(define (make-correspondence trip atlas)
  (if (null? trip)
      empty-correspondence
      (add-leg (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:

```
;: type: List<Segment>, List<Segment> -> List<Match<Segment, Segment>>

(define (make-correspondence trip atlas)
  (if (null? trip)
      empty-correspondence
      (let ((other (select atlas)))
        (add-leg (make-match (car trip) other)
                  (make-correspondence (cdr trip) 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.

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

Exercise 4: Complete the implementation for generating correspondences by providing the `select` 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.

To test this, temporarily define

```
(define remove-element (lambda (elt atlas) atlas))
```

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 `boston_travel.scm` into your Scheme environment.

Exercise 5: If you think about the above definition, you will realize that if we assume that Louis doesn't backtrack, then once we use a segment from the atlas, we don't need to use it again. So we would really like `remove-element` to return a new version of the atlas, with the selected leg removed. Thus, the `remove-element` procedure should take an element and an atlas (or 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` without using `delq`.

Show your new version of the 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 `boston-travel.scm` into your Scheme environment.

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

Exercise 6: 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 `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:

```
(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))
```

How many correspondences do you get in each case, and what is the maximum possible? What does this tell you about where you are?

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

Filtered matching

You saw that `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 `segment` data abstraction to support segments with unknown names. Extending the data abstraction requires adding a new constructor and predicate.

```
(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 "###")
```

The variable `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 `internal-name-for-unnamed-segment` from being used externally; for now, it is internal only by convention.

Exercise 7: Write a procedure called `same-name?`, which takes as argument a `match` containing two `segments` and which evaluates to true if the names of the two `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.

```
; same-name?: evaluates to true if the two segments have the same
; name or one is unnamed
;
; type: match<segment,segment> -> boolean
(define same-name? ...)
```

Exercise 8: Write a procedure `always?`, which takes a predicate (that is, a function that returns a boolean value) and a list of elements as input. `Always?` returns `true` if applying the `predicate` to each element in the list returns `true`, otherwise it returns `false`.

```
; always?: evaluates to true if a correspondence satisfies a
; predicate for each match
;
(define always? ...)

; for example
(always? integer? (list 1 2 3 4))
;Value: #t
```

```
(always? odd? (list 1 2 3 4))
;Value: #f
```

Exercise 9: Now, create a procedure `generate-constrained-correspondence-set`. This should look very much like `generate-correspondence-set`, the difference is that it also takes a `predicate` as argument, and applies that `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 previously, but now using the constrained version of the generator, and `same-name?` as the test. Note that since `same-name?` applies to matches, and we want our predicate to apply to correspondences, you will want to use `always?` as part of your predicate to apply `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 10: 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. This means that the Euclidean distance between the start and end points of the two segments are the same.

To do this, write a new predicate `same-name-and-length?` that, in addition to checking for `same-name?`, also compares the lengths of the trip-segment and atlas-segment.

Make sure to use appropriate selectors when accessing the endpoints and their coordinates.

Try out your new predicate by rerunning the experiments from Exercise 9, 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 11: 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.

```
; 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 12: 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 13: Try out your new predicate by rerunning the experiments from Exercise 9, 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 14 (open ended): Note that our tests have all really relied on a random testing of trip segments. A more realistic testing would be to simulate our process:

- given an atlas, pick an arbitrary translation and rotation, and generate a new version of the atlas, with each segment appropriately translated and rotated;
- pick an arbitrary starting point in this new version of the atlas;
- extend the trip by finding a new segment whose starting point matches the end point of the current segment;

- continue until either you run out of segments, or you reach some predefined length.

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

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.