

MAU11601

Lecture 02

2026-09-24

Positions

In the eclipse simulator the sun and moon moved only vertically, so we only need a single number to describe (the varying part of) their position.

Obviously we'd like to be able to describe two, or even three, dimensional motion.

The BSL comes with a composite type `posn` for 2-d positions.

To describe a position you need two numbers, so it comes with a function `make-posn` to get a `posn` from two numbers, and two functions `posn-x` and `posn-y` to extract those numbers from a `posn`.

For obvious reasons `make-posn` is called a constructor.

`posn-x` and `posn-y` are called either destructors or selectors. The word destructor mirrors constructor, but they don't destroy anything. `posn`'s are immutable.

If we type `(define P (make-posn 3 4))` in the definitions window and then `(posn-x P)` in the interaction window we get 3, as expected, but `P` still exists, and we can then type `(posn-y P)`, which gives 4.

The y part of a `posn` increases as we go down, not up!

More about positions

There's also a function `posn?` to tell you whether something is a `posn`. So `(posn? (make-posn 3 4))` returns `#true` while `(posn? 17)` returns `#false`. `#true` and `#false` are boolean values in BSL. In regular Racket they are `#t` and `#f`. Functions which return booleans are called predicates. By convention their names end with a `?`.

You can write functions which take `posn` arguments, like

```
(define (sq x) (* x x))  
(define (dist p q) (sqrt (+ (sq (- (posn-x p) (posn-x q)))  
                             (sq (- (posn-y p) (posn-y q))))))
```

Then `(dist (make-posn 3 4) (make-posn 0 0))` returns 5, as expected.

Defining the auxiliary function `sq` makes the definition of `dist` shorter and more readable. It's a good idea to look for repeated patterns and consider defining auxiliary functions for them.

Changing position

Consider a maze game. We use keyboard input to move the position of the character.

The state of the world is described by a posn.

We will need a function to take a key and update the position appropriately.

```
(define move-size 20)
(define (move p k)
  (cond
    [(key=? k "w") (make-posn (posn-x p) (- (posn-y p) move-size))]
    [(key=? k "s") (make-posn (posn-x p) (+ (posn-y p) move-size))]
    [(key=? k "a") (make-posn (- (posn-x p) move-size) (posn-y p))]
    [(key=? k "d") (make-posn (+ (posn-x p) move-size) (posn-y p))]
    [#true p]))
```

key=? is a predicate checking whether a key matches something. Remember that predicate names end by convention with a ?.

cond is a conditional. It checks conditions and expressions, evaluating the first condition whose expression evaluates to #true.

You can write else in place of #true for readability.

Drawing

As before, we need a starting state and a function to draw the current state of the world, which requires some preliminary definitions.

```
(define maze-size 400)
(define half-size (/ maze-size 2))
(define start (make-posn half-size half-size))
(define background (empty-scene maze-size maze-size))
(define (draw p)
  (place-image character-cat-girl1 (posn-x p) (posn-y p) background))
```

Images generally have copyright. You're going to have to publish your game so you need to obey copyright restrictions. PlanetCute have a library of copyright free images. `draw` should take the state of the world and give you the corresponding scene. Last time the state was described by a number. This time it's described by a `posn`. It could be anything, but your update functions need to take the old state and give you a new state, so in this case `move` takes a `posn` (and a key) and gives you a `posn`.

What's missing?

- Walls. Aren't mazes meant to have walls?
 - A goal. We need to go somewhere, and check whether we've arrived.
- These require more work, but they don't require any new language features.
- We need to add walls and a goal to our background.
- We need to modify `move` so it doesn't let you move through walls.
- We might want to check whether we're at the goal.
- We already have the tools to do all of these things, though maybe not efficiently.

Structure

A `posn` is an example of a structure, a composite datatype with named fields. The only thing which is special about it is that it comes predefined. You can define your own structures with `define-struct`.

Suppose we want a visual indicator of when we've finished. Now our state needs a position and a boolean, so we define `(define-struct world (place done?))`. This automatically creates a constructor `make-world` and selectors `world-place` and `world-done?`, and a predicate `world?`.

We expect the `x` and `y` fields of a `posn` both to be numbers, but this isn't enforced. We expect the type of the `place` field of a `world` to be a `posn` while the type of `done?` should be boolean, but again this isn't enforced.

Note that we have a structure within a structure. This is fine.

There are languages which require you to state the types of the fields in a structure, and then enforce this. For better or worse, Racket doesn't.

Consistency

Once you've changed the type of the world you have to change the initial state, the draw function, and the update functions that you give to big-bang, e.g,

```
(define start (make-world (make-posn half-size half-size) #false))  
(define (draw w)  
  (place-image character-cat-girl  
                (posn-x (world-place w))  
                (posn-y (world-place w))  
                (place-image (square maze-size 'solid  
                              (if (world-done? w)  
                                  'green  
                                  'red))  
                              half-size half-size background)))
```

Of course we need to update move as well.