Problem Statement

Determine a suitable representation for a snake in the world. Then set up a world such that the user can change the direction of the snake using the keyboard.

1 Setting up our data structures

The first step in our design is to come up with a data representation of our Snake. We decided that a snake should have two things: a direction, and a body.

> (struct snake (direction body))

The direction should be a string representing the key that a user has pressed, i.e. ‘left’ for the left arrow key. In this way, some later function can change the direction of a Snake by simply replacing its direction attribute with the key that was pressed.

> (define THE-SNAKE (snake "left" ...))

The body of the snake should be a list of segments, where the first segment represents the head of the snake. A segment consists of an x-y position and a sprite (an image which represents something), which will be used to render that segment properly.

> (struct segment (x y sprite))
> (define THE-SNAKE (snake "left"
  (list (segment 0 0 SEG-SPRITE)
        ...)))

When we create a struct, by default any instance of that struct will hide its contents from the world.
> (struct snake (direction body))
> (snake "left" 'idk)
#<snake>

However, by consulting the documentation we find that there is a keyword that can be used when creating a structure type that will allow us to see inside it.

> (struct snake (direction body)  
  #:transparent) ; make the contents of an instance of snake visible
> (snake "left" 'idk)
(snake "left" 'idk)

Now, when we have encountered structs before in the teaching languages, they came with a constructor function, i.e. make-student for a ‘student’ struct. But in Racket, the default constructor is simply the same name as the struct itself.

> (struct snake (direction body)  
  #:transparent) ; make the contents of an instance of snake visible
> (snake "left" 'idk); default constructor method
(snake "left" 'idk)
> (make-snake "left" 'idk)
ERROR: make-snake: undefined

If you wish to provide your own name for the constructor function, you can use another keyword to do this.\(^1\)

> (struct snake (direction body)  
  #:transparent ; make the contents of an instance of snake visible
  #:constructor-name make-snake); provide the constructor 'make-snake'
> (snake "left" 'idk)
ERROR: snake: bad syntax
> (make-snake "left" 'idk)
(snake "left" 'idk)

If for some reason you would like to have both methods of constructing your struct, use the keyword #:extra-constructor-name instead (do not use both keywords).

2 Learning to draw

Now that we’ve created representations for our data, we can move on to figuring out how they will be drawn.

\(^1\)See the documentation on structs: [http://docs.racket-lang.org/reference/define-struct.html](http://docs.racket-lang.org/reference/define-struct.html)
Figure 1: Rendering a snake.

> (render THE-SNAKE)
Our snake is made of segments, and each segment has an image associated with it. What we need is a function that will render all of these segments to the same image: an empty-scene. Here is an example of a recursive function that does just that.

> ; Renders a list of segments
(define render-segments
  (lambda (segments)
    (cond
      ((empty? segments) (empty-scene WORLD-W WORLD-H))
      (else (let ([s (first segments)])
               (place-image (segment-sprite s)
                            (segment-x s)
                            (segment-y s)
                            (render-segments (rest segments)))))))))

Our friend Mr. Nolan suggested a different method, using a built-in function.

> ; Renders a list of segments
(define nolan
  (lambda (segments)
    (foldl (lambda (s previous)
              (place-image (segment-sprite s)
                           (segment-x s)
                           (segment-y s)
                           previous))
           (empty-scene WORLD-W WORLD-H)
           segments))))

Using either of these methods produces the same output for a given snake, shown in Figure 1.

3 And it all started with the big-bang...

We now have all the tools we need to start the big-bang: we have a snake, and a way to draw it. Setting up a big-bang such that it draws the same snake on every tick is simple, and will produce a very similar result to what you saw in Figure 1.

> (big-bang THE-SNAKE
  (on-tick (lambda (world) world) ; Don’t change the world.
           1/2) ; Tick every 1/2 second.
  (to-draw render)) ; Render the snake on every tick.
But as satisfied as you might feel about having accomplished this feat, the fact remains that it just isn’t very interesting.

So let’s make it move.

First, we need a function that will handle keyboard events.

> (define change-direction
  (lambda (world key)
    ...
  )

This event-handler takes the current world and the key that was pressed. But remember, our world is just a snake! There is nothing else in our world right now except a single snake. So let’s give it a better name.²

> (define change-direction
  (lambda (snake key)
    ...
  ))

Back when we were designing our data structures, we said that the direction attribute of a snake would correspond exactly to the key that represents that direction. So this function is simple: return a new snake with the key that was pressed as its direction, and the same body as the snake that was given.

> (define change-direction
  (lambda (snake key)
    (make-snake key (snake-body snake))
  ))

If you were paying close attention, you may have picked up on a bug in this event-handler: we want to change direction based on key presses, but only on specific key presses! Not any old key! So let’s put a constraint on this function.³

> (define change-direction
  (lambda (snake key)
    (if (member key VALID-KEYS)
      (make-snake key (snake-body snake))
      snake) ) ; Return the original snake if the key was invalid

Sweet. Now we can write a big-bang that changes the direction of our snake on specific key-presses!

²Here is where specifying a constructor in the definition of the snake struct is advantageous: we can use the variable ‘snake’ without losing access to the snake constructor.

³See the documentation for ‘member’ vs. ‘memq’ vs. ‘memv’ in Racket.
But wait... the snake still isn’t moving! Our function must be broken. To check this, add a print statement to the function in the on-tick clause, and you will be able to see the direction of the snake and watch it respond to key presses.

Well, the direction is certainly changing! What’s going on? The problem is that though we defined a function which changes the direction of the snake, we didn’t design one that actually *updates* the image being drawn.

### 4 Re-learning to draw

So what exactly is it that we need to do? We need to take a snake, like the one in Figure 2, and return a new snake just like it, but with the segments in different positions. These positions will be based on the direction the snake is *currently* heading. Our *change-direction* function will keep the *direction* attribute of our snake up to date, and our *update-snake* function will make sure the position of the snake’s body reflects this attribute.

This may seem like a daunting task: go through the body of the snake, moving the head in the proper direction and giving each subsequent segment the position of the previous segment, thereby shifting the segments in the proper direction. But if you look closely at the two diagrams, you will notice that there might be an easier way: the same thing could be accomplished by simply chopping off the tail (the last segment) and attaching a new head to the front of the body. So let’s do that.
> (define update-snake
  (lambda (snake)
    (cond
      [(equal? (snake-direction snake) "left") ; Move the snake left.
        (make-snake ... ...)]
      [(equal? (snake-direction snake) "right") ; Move the snake right.
        ...]
      [(equal? (snake-direction snake) "up") ; Move the snake up.
        ...]
      [(equal? (snake-direction snake) "down") ; Move the snake down.
        ...]
      [else snake]])) ; In case something goes wrong, return the snake unchanged.

Notice that I have decided that the snake will respond to the arrow keys. You could have chosen any set of keys you want (be sure they match what you put in your VALID-KEYS definition), like the traditional W-A-S-D set for example.

Now comes the clever part. Let’s work on the first branch of the cond first. We know that we will always be returning a snake; that much is easy.

> (define update-snake
  (lambda (snake)
    (cond
      [(equal? (snake-direction snake) "left") ; Move the snake left.
        (make-snake "left" ... ...)]
      [(equal? (snake-direction snake) "right") ...] ; Move the snake right.
      [(equal? (snake-direction snake) "up") ...] ; Move the snake up.
      [(equal? (snake-direction snake) "down") ...] ; Move the snake down.
      [else snake]])) ; In case something goes wrong, return the snake unchanged.

We also know that the snake we return will maintain its current direction because remember, this function is only concerned with making the body of the snake match its direction, not with changing the direction.

> (define update-snake
  (lambda (snake)
    (cond
      [(equal? (snake-direction snake) "left") ; Move the snake left.
        (make-snake "left" ... ...)]
      [(equal? (snake-direction snake) "right") ...] ; Move the snake right.
      [(equal? (snake-direction snake) "up") ...] ; Move the snake up.
      [(equal? (snake-direction snake) "down") ...] ; Move the snake down.
      [else snake]])) ; In case something goes wrong, return the snake unchanged.

Now we need to make the body of the snake move to the left. But how far? This is an important question to ask ourselves: how far should the snake move on each tick? Well, we know it should be a fixed amount: the user will never be in control of how far the snake
is moving or how fast, only which direction it moves in.

So let’s imagine the scene as a grid, and each segment of the snake occupies one cell in that grid, just like in Figure 3. The width of each cell is the width of a snake segment, so that’s how far the snake should move. And it should move that far in whatever direction it is heading. Using our shortcut method, the only segment that needs moving is the head of the snake. So when it is moving left, for example, it’s x-coordinate should be decreased by the width of a snake segment, while it’s y-coordinate stays the same.

To do this, we create a new head with the changed x-coordinate, the same y-coordinate, and same sprite as the old head.

> (define update-snake
  (lambda (snake)
    (cond
      [(equal? (snake-direction snake) "left") ; Move the snake left.
       (make-snake "left" (cons (make-segment (- (segment-x (first (snake-body snake))) SEGMENT-W) (segment-y (first (snake-body snake)))) HEAD-SPRITE) ; Heads have a special sprite.
        ...))]
      [(equal? (snake-direction snake) "right") ...] ; Move the snake right.
      [(equal? (snake-direction snake) "up") ...] ; Move the snake up.
      [(equal? (snake-direction snake) "down") ...] ; Move the snake down.
      [else snake]])) ; In case something goes wrong, return the snake unchanged.

Now that we’ve created a new head, we need to attach it to the old body, but the old body with the tail (the last segment) removed. To remove the tail, we can just take the rest of the reversed list, and then reverse it again to put it back in the right order.

> (define update-snake
  (lambda (snake)
    (cond
      [(equal? (snake-direction snake) "left") ; Move the snake left.
       (make-snake "left" (cons (make-segment (- (segment-x (first (snake-body snake))) SEGMENT-W) (segment-y (first (snake-body snake)))) HEAD-SPRITE) ; Heads have a special sprite.
        (reverse (rest (reverse (snake-body snake)))))
        ...))]
      [(equal? (snake-direction snake) "right") ...] ; Move the snake right.
      [(equal? (snake-direction snake) "up") ...] ; Move the snake up.
      [(equal? (snake-direction snake) "down") ...] ; Move the snake down.
      [else snake]])) ; In case something goes wrong, return the snake unchanged.
Phew! Look at that mess of code we’ve made. But at least you understand what it’s doing! Personally, I dislike ugly code, and prefer to make it as readable as possible, when I can, and comment heavily when I can’t. Here, I would do some cleaning up (especially since I will likely be doing similar calculations in the rest of the conditional branches), using \texttt{let}.

\begin{verbatim}> (define update-snake
  (lambda (snake)
    (let* ([dir (snake-direction snake)]
           [body (snake-body snake)]
           [head (first body)]
           [new-body (reverse (rest (reverse body)))]
      (cond
        [(equal? dir "left") ; Move the snake left.
         (make-snake dir (cons (make-segment (- (segment-x head) SEGMENT-W) ; New x
                                 (segment-y head) ; Old y
                                 HEAD-SPRITE) ; Head sprite
                        new-body))]
        [(equal? dir "right") ...] ; Move the snake right.
        [(equal? dir "up") ...] ; Move the snake up.
        [(equal? dir "down") ...] ; Move the snake down.
        [else snake]])) ; In case something goes wrong, return the snake unchanged.
  
Even just that small bit of clean-up makes it look more readable.

5 Back to the big-bang

Alright, we now have a movable snake! Let’s set up a \texttt{big-bang} that let’s us play with our creation.

\begin{verbatim}> (big-bang THE-SNAKE
  (on-tick update-snake ; Update the snake on every tick.
                       1/2) ; Tick every 1/2 second.
  (on-key change-direction) ; Change direction on key press.
  (to-draw render)) ; Render the snake on every tick.
  
It works! It’s beautiful! But... wait, what is happening?! The snake is turning blue! It moves wonderfully, but it has turned completely blue. What is going on?!

The problem is that when we created a new head and stuck it on the old body, we neglected to change the sprite of the old head to that of a normal segment. Because of this, the first time \texttt{update-snake} was called, it just added a second head to the snake and removed a

\footnote{Notice I use the starred version in the code; it allows me to reference previous bindings in the same \texttt{let}.}
segment from the end, and every subsequent call did the same thing until all that was left was a body made entirely out of heads! To remedy this, all we have to do is modify new-body so that it changes the sprite of the old head to that of a normal segment.

> (define update-snake
  (lambda (snake)
    (let* ([dir (snake-direction snake)]
           [body (snake-body snake)]
           [head (first body)]
           [new-body (cons (make-segment (segment-x head)
                                           (segment-y head)
                                           SEGMENT-SPRITE) ; Modify the old head sprite
                         (rest (reverse (rest (reverse body)))))]) ; Get rest of body
      (cond
        [(equal? dir "left") ; Move the snake left.
         (make-snake dir (cons (make-segment (- (segment-x head) SEGMENT-W) ; New x
                                      (segment-y head) ; Old y
                                      HEAD-SPRITE) ; Head sprite
                       new-body))]
        [(equal? dir "right") ...] ; Move the snake right.
        [(equal? dir "up") ...] ; Move the snake up.
        [(equal? dir "down") ...] ; Move the snake down.
        [else snake]])))) ; In case something goes wrong, return the snake unchanged.

And there you have it: stage 1 complete!

You may have noticed that our snake is allowed to explore the area outside the visible scene; take a look at how we corrected this issue in our Homework Five and see if you can apply a similar logic here. You should also try to make a snake generator, so that you can create random snakes to use in your game.

Happy coding!
Figure 2: A snake, heading down, with a grid showing its individual segments.

Figure 3: The previous snake, now heading left.