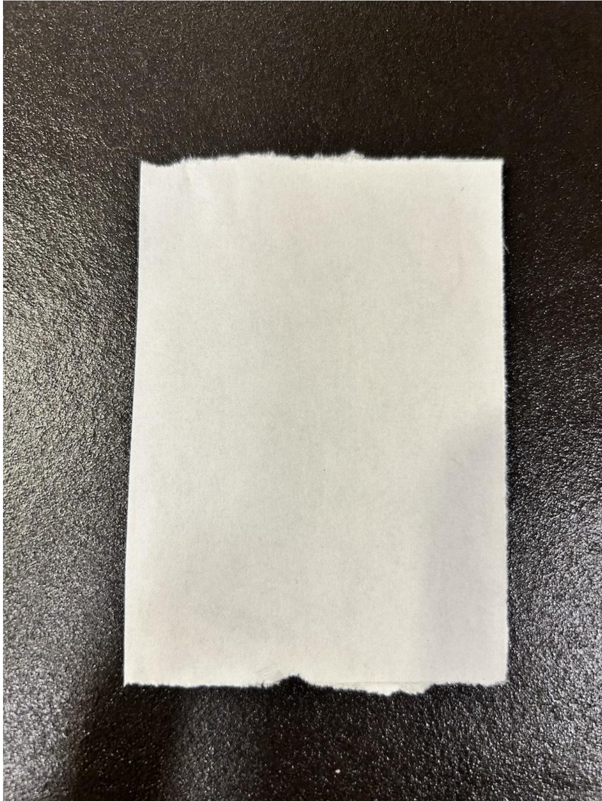
# Lists

CS135 Lecture 05

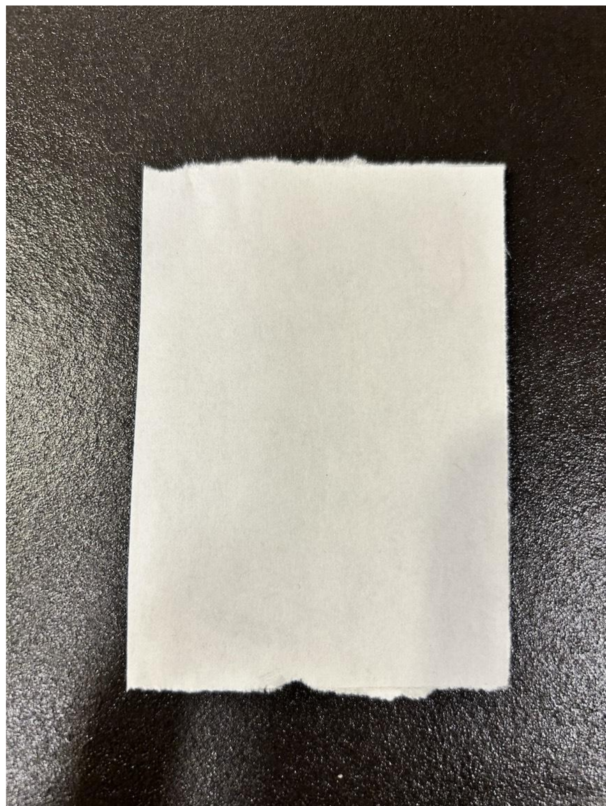
# L05.0 List values and expressions

# This is an empty list



Since it’s empty, we don’t know what kind of list it is. It might be a grocery list. It might be a list of things to do.

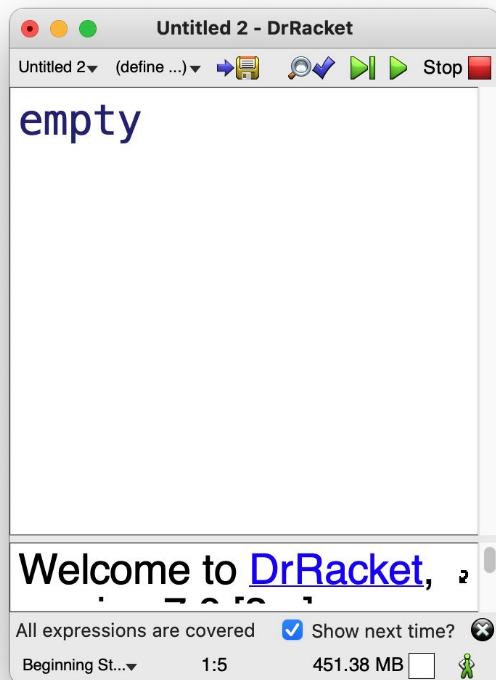
# Having an empty list is not the same as having no list

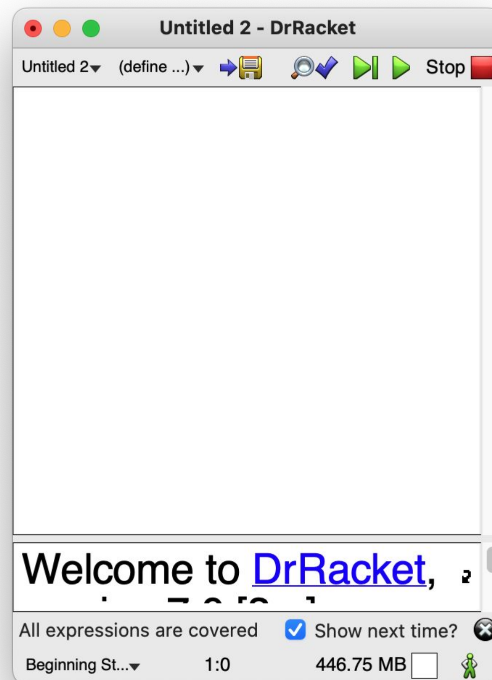


# This is an empty list in Racket

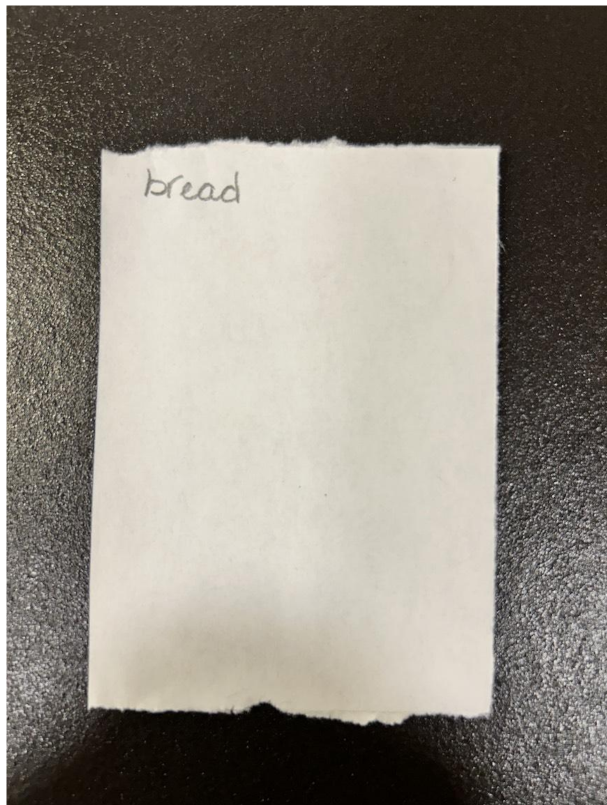
Since it's empty, we don't know what kind of list it is. It might be list of `Int`. It might be a list of `Sym`.

# Having an empty list is not the same as having no list



# Let’s add an item to our list




Looks like a grocery list.

# Let's add an item to our Racket list with `cons`

Untitled 2 - DrRacket

Untitled 2 (define ...) Check Syntax Step Run Stop

```
(cons 'bread empty)
```

```
>
```

All expressions are covered — Show next time?

Beginning Student custom — 4:2 — 373.47 MB

Looks like a list of **`Sym`**.

**`cons`** constructs a list by adding an item to the front of another list, e.g. **`empty`**.

**`cons`** can be confusing because it can be viewed as a function or a way of representing the resulting value.

# Let’s add another item to our list

Untitled 2 - DrRacket

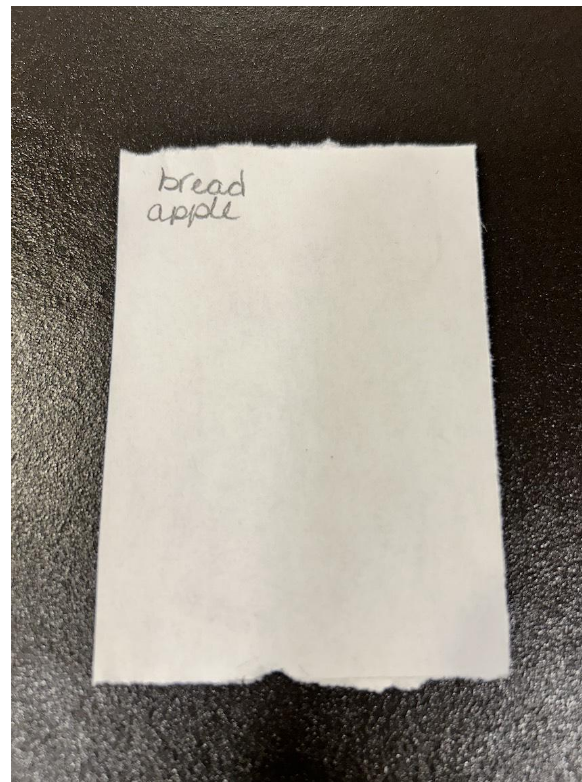
Untitled 2 (define ...) Check Syntax Step Run Stop

```
(cons 'apple
      (cons 'bread empty))
```

```
>
```

All expressions are covered Show next time?

Beginning Student custom 4:2 403.50 MB



# Let’s add a third item

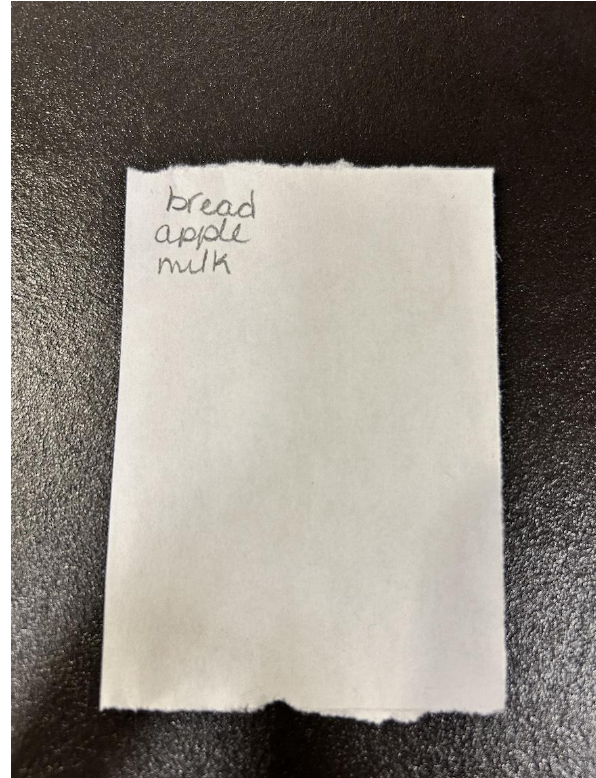
Untitled 2 - DrRacket

```
(cons 'milk
      (cons 'apple
            (cons 'bread empty)))
```

```
>
```

All expressions are covered

Beginning Student custom



# We have milk in the fridge. Let’s erase it.

Untitled 2 - DrRacket

Untitled 2 (define ...) Check Syntax Step Run Stop

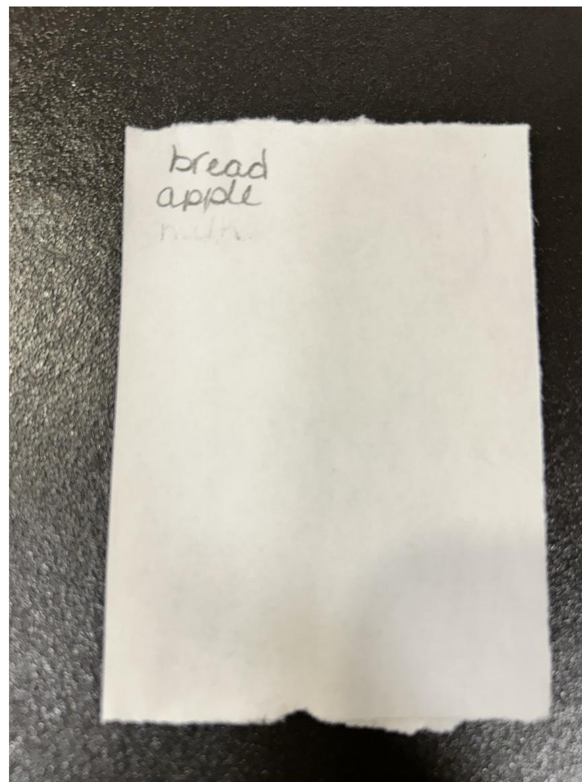
```
(rest (cons 'milk
            (cons 'apple
                  (cons 'bread empty))))
```

Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.

```
(cons 'apple (cons 'bread empty))
>
```

All expressions are covered Show next time?

Beginning Student custom 4:2 494.98 MB



# The `rest` function

Untitled 2 - DrRacket

Untitled 2 (define ...) Check Syntax Step Run Stop

```
(rest (cons 'milk
            (cons 'apple
                  (cons 'bread empty))))
```

Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.

```
(cons 'apple (cons 'bread empty))
>
```

All expressions are covered   Show next time?
Beginning Student custom   4:2   494.98 MB

`rest` is a racket function that consumes a list and produces that list with the first item removed.

It is an error to apply `rest` to the `empty` list.

# The `first` function

Untitled 2 - DrRacket

Untitled 2 (define ...) Check Syntax Step Run Stop

```
(first (cons 'milk
             (cons 'apple
                   (cons 'bread empty))))
```

Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.

```
'milk
>
```

All expressions are covered Show next time?

Beginning Student custom 4:2 538.48 MB

**`first`** is a racket function that consumes a list and produces the first item of that list.

It is an error to apply **`first`** to the **`empty`** list.

# Add some eggs

Untitled 2 - DrRacket

Untitled 2 (define ...) Check Syntax Step Run Stop

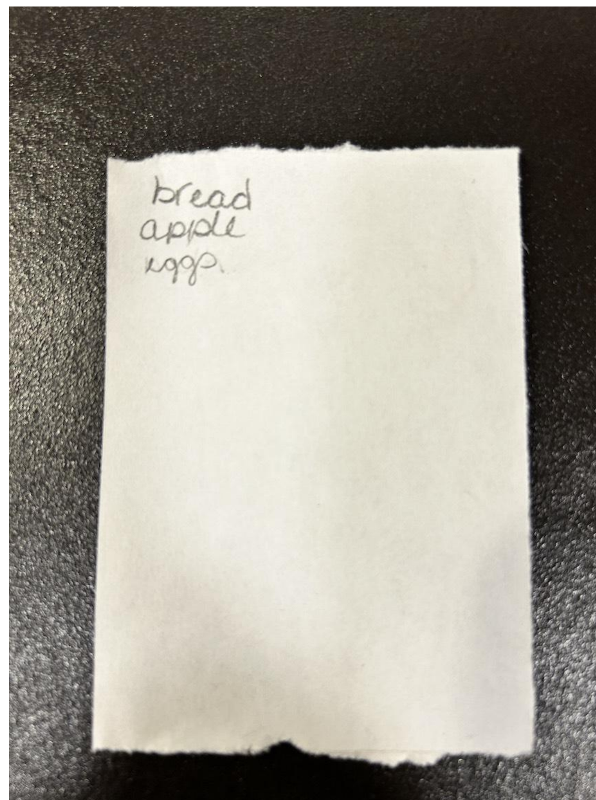
```
(cons 'eggs
      (rest (cons 'milk
            (cons 'apple
                  (cons 'bread empty)))))
```

Language: Beginning Student [custom]; memory limit: 512 MB.

```
(cons 'eggs (cons 'apple (cons 'bread
empty)))
>
```

All expressions are covered  Show next time?

Beginning Student custom  4:2  337.46 MB



# Another apple

Untitled 2 - DrRacket

Untitled 2 (define ...) Check Syntax Step Run Stop

```
(cons 'apple
    (cons 'eggs
        (rest
            (cons 'milk
                (cons 'apple
                    (cons 'bread empty))))))
```

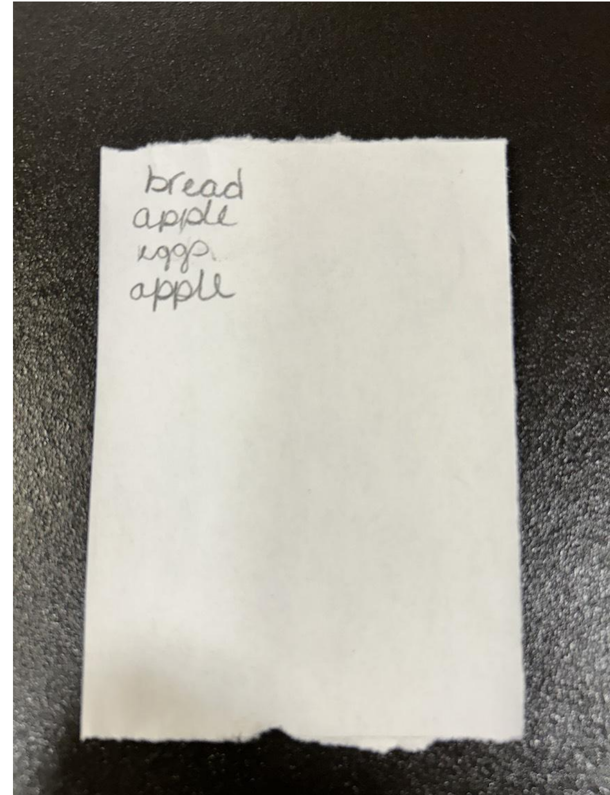
Language: Beginning Student [custom]; memory limit: 512 MB.

```
(cons 'apple (cons 'eggs (cons 'apple
(cons 'bread empty))))
>
```

All expressions are covered

Show next time?

Beginning Student custom 4:2 417.80 MB



# Lists are values

Untitled 2 - DrRacket

Untitled 2 (define ...) Check Syntax Step Run Stop

```
(cons 'apple
      (cons 'eggs
            (rest
             (cons 'milk
                   (cons 'apple
                         (cons 'bread empty))))))
```

```
Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.
(cons 'apple (cons 'eggs (cons 'apple (cons 'bread empty))))
>
```

All expressions are covered Show next time?

Beginning Student custom 4:2 492.55 MB

The definitions pane on the top contains an expression (because of **`rest`**).

The interactions pane on the bottom contains a value.

Lists are the central data structure we use in CS135.

# Testing for the empty list with **`empty?`**

Untitled 2 - DrRacket

Untitled 2 (define ...) Check Syntax Step Run Stop

```
(empty? (cons 'apple (cons 'bread empty)))
(empty? (rest (cons 'milk empty)))
(empty? empty)
(empty? 123)
```

```
false
true
true
false
>
```

All expressions are covered  Show next time?

Beginning Student custom  7:2  485.98 MB

**`empty?`** consumes <u>any value</u> and produces **`true`** only if it is the empty list

# List of Racket list operations

| | |
|---|---|
| **`cons`** | <u>Cons</u>tructs a list from a value and a list by adding the value to the front. |
| **`first`** | Consumes a non-empty list and produces the first value in that list. |
| **`rest`** | Consumes a non-empty list and produces a list with the first value removed. |
| **`empty?`** | Consumes <u>any value</u> and produces true only if the value is **`empty`**. |
| **`list?`** | Consumes <u>any value</u> and produces true only if the value is list. |
| **`cons?`** | Consumes <u>any value</u> and produces true only if the value is a non-empty list. |

# L05.1 Composite data

# Composite data

Now that we have lists, we can create data types that are more than just a single number or symbol, i.e., *composite* data types

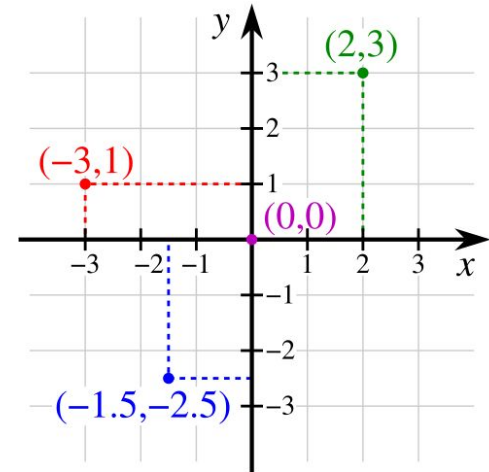
For example, we could use a list of two Num to represent a point in the Cartesian coordinate system: (*x*, *y*).

We represent the point (-3, 1) as:

```
(cons -3 (cons 1 empty))
```

More generally, we represent the point (*x*, *y*) as:

```
(cons x (cons y empty)
```



# Distance from the origin

We want to write a function the computes the distance from a point (*x*, *y*) to the origin (0, 0).

This is the first draft of our **purpose**.

From our math classes we know that the distance from (*x*, *y*) to (0, 0) is $\sqrt{x^2 + y^2}$.

In thinking of **examples**, we want some with *x* positive, some with *x* negative, some *y* positive, etc., as well as some with *x* and/or *y* zero.

$$(3, -1) \Rightarrow \sqrt{3^2 + -1^2} \approx 3.1622$$

$$(6, 0) \Rightarrow \sqrt{6^2 + 0^2} = 6$$

$$(-3, 4) \Rightarrow \sqrt{-3^2 + 4^2} = 5$$

$$(0, 0) \Rightarrow \sqrt{0^2 + 0^2} = 0$$

# Header

Let’s give a name to our function.

One possibility is **`distance-to-origin`**, which is accurate but long, with lots of typing. Too long can be confusing in a larger context with lots of functions.

On the other hand, **`d0`**, is short but too cryptic. Let’s err on the side of long.

```
(define (distance-to-origin point)...)
```

# Contract

Our function consumes a point and produces a **`Num`**.

A point is a **`(cons x (cons y empty))`** so we can write our **contract** as:

```
;; distance-to-origin: (cons x (cons y empty)) -> Num
```

We can now finalize our **purpose** as:

```
;; distance-to-origin consumes a point in the Cartesian
;; coordinate system and produces the distance to the origin.
```

# Body

At this point, we understand our problem fairly well, and we have a good idea how the data will be represented in Racket.

Since there's no recursion, we don't use the template. We can just translate the math directly into Racket.

```
(define (distance-to-origin point)
  (sqrt (+
         (sqr (first point)) ; get x from the point
         (sqr (first (rest point))) ; get y from the point
         )))
```

We've added some comments since the access to *x* and *y* seems confusing.

# Putting it all together

```
;; distance-to-origin consumes a point in the Cartesian
;; coordinate system and produces the distance to the origin.
;; distance-to-origin: (cons x (cons y empty)) -> Num
(define (distance-to-origin point)
  (sqrt (+
         (sqr (first point)) ; get x from the point
         (sqr (first (rest point))) ; get y from the point
         )))
(check-expect (distance-to-origin (cons -3 (cons 4 empty))) 5)
(check-within
  (distance-to-origin (cons 3 (cons -1 empty))) 3.1622 0.001)
(check-expect (distance-to-origin (cons 6 (cons 0 empty))) 6)
(check-expect (distance-to-origin (cons 0 (cons 0 empty))) 0)
```

# L05.2 Data definitions

# Data types

We use various types in our contracts to help document the behaviour of our functions.

These types include **`Sym`**, **`Nat`**, **`Rat`**, etc.

The contract for **`distance-to-origin`** may be hard to understand because the data type it consumes is composite.

```
;; distance-to-origin: (cons x (cons y empty)) -> Num
```

# Data definitions

We can use a *data definition* to give a name to a composite data type.

```
;; a Point is a (x,y) point in the Cartesian coordinate system
;; a Point is a (cons Num (cons Num empty))
```

With this data definition, we can simplify our contract for `distance-to-origin`.

```
;; distance-to-origin consumes a point in the Cartesian
;; coordinate system and produces the distance to the origin.
;; distance-to-origin: Point -> Num
```

# Using helper functions

To make things more understandable, we can also create *helper functions* to create a Point and to access its components.

```
;; a Point is a point in the Cartesian coordinate system
;; a Point is a (cons x (cons y empty)
;;
;; mk-point consumes an x and y coordinate and produces a Point
;; mk-point: Num Num -> Point
(define (mk-point x y) (cons x (cons y empty)))

;; get-x consumes a Point and produces its x coordinate
;; get-x: Point -> Num
(define (get-x point) (first point))

;; get-y consumes a Point and produces its y coordinate
;; get-y: Point -> Num
(define (get-y point) (first (rest point)))
```

# Using helper functions

```
;; distance-to-origin consumes a point in the Cartesian
;; coordinate system and produces the distance to the origin.
;; distance-to-origin: Point -> Num
(define (distance-to-origin point)
  (sqrt (+ (sqr (get-x point)) (sqr (get-y point)))))

(check-expect (distance-to-origin (mk-point 3 4)) 5)
(check-within
  (distance-to-origin (mk-point 3 -1)) 3.1622 0.001)
(check-expect (distance-to-origin (mk-point 6 0)) 6)
(check-expect (distance-to-origin (mk-point 0 0)) 0)
```

# A note on structures

DrRacket supports a feature called “structures”, which are composite data types similar to the lists in the previous slides. You may see structures mentioned in DrRacket documentation and in previous iterations of CS135.

On the one hand, structures do all the work of creating helper functions. Defining a structure automatically creates functions to assess its components.

On the other hand, lists are much more powerful than structures. Anything you can do with a structure, you can do with a list of fixed size.

In CS135, we have only one composite data type, the list. Almost, anyway. As you will see next lecture, data definitions for lists can be recursive, allowing us to work with composite data of arbitrary size.

# Data definitions

We can also create data definitions to give names to sets of symbols a function might produce or consume.

```
;; an Outerwear is (AnyOf 'jacket, 'sweater, 'shirt)

;; what-to-wear: Num -> Outerwear
(define (what-to-wear temperature)
  (cond [(< temperature 8) 'jacket]
        [(< temperature 16) 'sweater]
        [else 'shirt]))
```

# Lecture 05 Summary

# L05: You should know

- How to create and manipulate lists with `cons`, `first`, `rest`, `empty`, `empty?`, `list?`, and `cons?`
- How to write data definitions and helper functions for composite data types using lists.
- How to apply the design pattern to create functions that work with composite data types using lists.

# L05: Allowed constructs

Newly allowed constructs:
**`cons cons? empty empty? first list? rest`**

Previously allowed constructs:
```
( ) [ ] + - * / = < > <= >= ;
abs acos and asin atan check-expect check-within cond cos
define e else exp expt false inexact? log max min not
number? or pi quotient remainder sin sqr sqrt sub1 symbol?
symbol=? tan true zero?
AnyOf Bool Int Nat Num Rat Sym
```

Recursion **must** follow the Rules of Recursion (first version)