Structures and All That

September 26, 2019

"Here at Brymar College
We can get you prepared for the 31st century
With advanced programming and quad rendering
And Java plus plus plus scripting language
We offer advanced job placement assistance"

from Upgrade by Deltron 3030

Data Descriptions Matter

We have taken a weird approach by fixating on data structured in the form of "or" first.

Data Descriptions Matter

We have taken a weird approach by fixating on data structured in the form of “or” first.

- We would say that a traffic light’s state is red **or** yellow **or** green. I will sometimes refer to data in this form as defining a *sum type*, but for now I will stick to saying itemization.

Data Descriptions Matter

We have taken a weird approach by fixating on data structured in the form of “or” first.

- We would say that a traffic light’s state is red **or** yellow **or** green. I will sometimes refer to data in this form as defining a *sum type*, but for now I will stick to saying itemization.
- But tons of data is written in a compound manner. A person has a head **and** a face **and** a body … I will sometimes refer to data in this form as being a *product type*, but will usually stick to saying *struct*. When I talk about classes, I will be talking about more than compound data.

Data Descriptions Matter

We have taken a weird approach by fixating on data structured in the form of “or” first.

- We would say that a traffic light’s state is red **or** yellow **or** green. I will sometimes refer to data in this form as defining a *sum type*, but for now I will stick to saying itemization.
- But tons of data is written in a compound manner. A person has a head **and** a face **and** a body … I will sometimes refer to data in this form as being a *product type*, but will usually stick to saying *struct*. When I talk about classes, I will be talking about more than compound data.
- Whereas with “or” we would check which kind of data we would have and then use a computation specific to that data, with products we can directly project out data.

Data Descriptions Matter

We have taken a weird approach by fixating on data structured in the form of "or" first.

- We would say that a traffic light's state is red **or** yellow **or** green. I will sometimes refer to data in this form as defining a *sum type*, but for now I will stick to saying itemization.
- But tons of data is written in a compound manner. A person has a head **and** a face **and** a body ... I will sometimes refer to data in this form as being a *product type*, but will usually stick to saying *struct*. When I talk about classes, I will be talking about more than compound data.
- Whereas with "or" we would check which kind of data we would have and then use a computation specific to that data, with products we can directly project out data.
- Let's say that in Java that you have some person class with a first and last name represented as strings.

Data Descriptions Matter

We have taken a weird approach by fixating on data structured in the form of "or" first.

- We would say that a traffic light's state is red **or** yellow **or** green. I will sometimes refer to data in this form as defining a *sum type*, but for now I will stick to saying itemization.
- But tons of data is written in a compound manner. A person has a head **and** a face **and** a body ... I will sometimes refer to data in this form as being a *product type*, but will usually stick to saying *struct*. When I talk about classes, I will be talking about more than compound data.
- Whereas with "or" we would check which kind of data we would have and then use a computation specific to that data, with products we can directly project out data.
- Let's say that in Java that you have some person class with a first and last name represented as strings.
- It is easy to define a method that returns the person's full name by concatenating the first and last name.

Who Needs Structs Anyway?

So, why do we need compound data?

Who Needs Structs Anyway?

So, why do we need compound data?

- The obvious answer is that we have programs that have some kind of compound state.

Who Needs Structs Anyway?

So, why do we need compound data?

- The obvious answer is that we have programs that have some kind of compound state.
- Consider the simple program where we wanted to move a dot left and right.

Who Needs Structs Anyway?

So, why do we need compound data?

- The obvious answer is that we have programs that have some kind of compound state.
- Consider the simple program where we wanted to move a dot left and right.
- We were able to represent the state of the world as a single position number.

Who Needs Structs Anyway?

So, why do we need compound data?

- The obvious answer is that we have programs that have some kind of compound state.
- Consider the simple program where we wanted to move a dot left and right.
- We were able to represent the state of the world as a single position number.
- Let's add another dimension of movement where we can now move the dot up and down.

Who Needs Structs Anyway?

So, why do we need compound data?

- The obvious answer is that we have programs that have some kind of compound state.
- Consider the simple program where we wanted to move a dot left and right.
- We were able to represent the state of the world as a single position number.
- Let's add another dimension of movement where we can now move the dot up and down.
- Can we represent the state of the world as a single number?

Who Needs Structs Anyway?

So, why do we need compound data?

- The obvious answer is that we have programs that have some kind of compound state.
- Consider the simple program where we wanted to move a dot left and right.
- We were able to represent the state of the world as a single position number.
- Let's add another dimension of movement where we can now move the dot up and down.
- Can we represent the state of the world as a single number?
- If you said no, I get it! But that happens to be incorrect.

Who Needs Structs Anyway?

So, why do we need compound data?

- The obvious answer is that we have programs that have some kind of compound state.
- Consider the simple program where we wanted to move a dot left and right.
- We were able to represent the state of the world as a single position number.
- Let's add another dimension of movement where we can now move the dot up and down.
- Can we represent the state of the world as a single number?
- If you said no, I get it! But that happens to be incorrect.
- We can represent a grid with one number in the same sense that we can simulate a 10x10 2D array with a 100 element array.

Structs Make Things Easier

Personally, I like doing things the easy way.

Unstructured Compound Data?

We can actually represent compound data without needing to provide names for the individual pieces of data.

Unstructured Compound Data?

We can actually represent compound data without needing to provide names for the individual pieces of data.

- Let's consider some individual examples in Python.

Unstructured Compound Data?

We can actually represent compound data without needing to provide names for the individual pieces of data.

- Let's consider some individual examples in Python.
- We can represent a Person with a first name, last name, and age with the following kind of tuple:

Unstructured Compound Data?

We can actually represent compound data without needing to provide names for the individual pieces of data.

- Let's consider some individual examples in Python.
- We can represent a Person with a first name, last name, and age with the following kind of tuple:
- `("Peter", "Campora", 26)`

Unstructured Compound Data?

We can actually represent compound data without needing to provide names for the individual pieces of data.

- Let's consider some individual examples in Python.
- We can represent a Person with a first name, last name, and age with the following kind of tuple:
- `("Peter", "Campora", 26)`
- We could then define functions to act like field accesses.

Unstructured Compound Data?

We can actually represent compound data without needing to provide names for the individual pieces of data.

- Let's consider some individual examples in Python.
- We can represent a Person with a first name, last name, and age with the following kind of tuple:
- `("Peter", "Campora", 26)`
- We could then define functions to act like field accesses.

```
def first_name(tup):
  return tup[0]
```

-

Data (Un)Structures

We can actually define other data structures in terms of things like lists.

Data (Un)Structures

We can actually define other data structures in terms of things like lists.

- Let's consider defining a binary tree in terms of a python list.

Data (Un)Structures

We can actually define other data structures in terms of things like lists.

- Let's consider defining a binary tree in terms of a python list.
- We can define a null node with `[]`

Data (Un)Structures

We can actually define other data structures in terms of things like lists.

- Let's consider defining a binary tree in terms of a python list.
- We can define a null node with `[]`
- A tree with a single root element can be `[[] 1 []]`

Data (Un)Structures

We can actually define other data structures in terms of things like lists.

- Let's consider defining a binary tree in terms of a python list.
- We can define a null node with `[]`
- A tree with a single root element can be `[[] 1 []]`
- Here's a nice balanced tree `[[[] 1 []] 2 [[] 3 []]]`

Data (Un)Structures

We can actually define other data structures in terms of things like lists.

- Let's consider defining a binary tree in terms of a python list.
- We can define a null node with `[]`
- A tree with a single root element can be `[[] 1 []]`
- Here's a nice balanced tree `[[[] 1 []] 2 [[] 3 []]]`
- This representation gets a bit ugly fast, huh?

Data (Un)Structures

We can actually define other data structures in terms of things like lists.

- Let's consider defining a binary tree in terms of a python list.
- We can define a null node with `[]`
- A tree with a single root element can be `[[] 1 []]`
- Here's a nice balanced tree `[[[] 1 []] 2 [[] 3 []]]`
- This representation gets a bit ugly fast, huh?
- So Python gives classes (or named tuples) as a way to more easily define such structured data.

Unstructured Data in Racket

Similarly, we can define data using pairs and lists in Racket.

Unstructured Data in Racket

Similarly, we can define data using pairs and lists in Racket.

- We can write a pair $(1, 2)$ as `(1 . 2)`.

Unstructured Data in Racket

Similarly, we can define data using pairs and lists in Racket.

- We can write a pair $(1, 2)$ as `(1 . 2)`.
- We can write a linked list 1→2-→3→4→empty as `'(1 2 3 4)`

Unstructured Data in Racket

Similarly, we can define data using pairs and lists in Racket.

- We can write a pair $(1, 2)$ as `(1 . 2)`.
- We can write a linked list 1→2-→3→4→empty as `'(1 2 3 4)`
- To get the first element in the linked list, you can write: `(first '(1 2 3 4))`$\hookrightarrow$ `1`

Unstructured Data in Racket

Similarly, we can define data using pairs and lists in Racket.

- We can write a pair $(1, 2)$ as `(1 . 2)`.
- We can write a linked list 1→2-→3→4→empty as `'(1 2 3 4)`
- To get the first element in the linked list, you can write: `(first '(1 2 3 4))`$\hookrightarrow$ `1`
- To get the rest of the linked list you can write `(rest '(1 2 3 4))`$\hookrightarrow$ `'(2 3 4)`

Unstructured Data in Racket

Similarly, we can define data using pairs and lists in Racket.

- We can write a pair $(1, 2)$ as `(1 . 2)`.
- We can write a linked list 1→2-→3→4→empty as `'(1 2 3 4)`
- To get the first element in the linked list, you can write: `(first '(1 2 3 4))`$\hookrightarrow$ `1`
- To get the rest of the linked list you can write `(rest '(1 2 3 4))`$\hookrightarrow$ `'(2 3 4)`
- The empty list is represented with `'()` and you can check for the empty list with `(empty? '())` $\hookrightarrow$ `\#t`

Unstructured Data in Racket

Similarly, we can define data using pairs and lists in Racket.

- We can write a pair $(1, 2)$ as `(1 . 2)`.
- We can write a linked list 1→2-→3→4→empty as `'(1 2 3 4)`
- To get the first element in the linked list, you can write: `(first '(1 2 3 4))`$\hookrightarrow$ `1`
- To get the rest of the linked list you can write `(rest '(1 2 3 4))`$\hookrightarrow$ `'(2 3 4)`
- The empty list is represented with `'()` and you can check for the empty list with `(empty? '())` $\hookrightarrow$ `\#t`
- We will return to discussing lists in more detail later, since they are *extremely* important.

Unstructured Data in Racket

Similarly, we can define data using pairs and lists in Racket.

- We can write a pair $(1, 2)$ as `(1 . 2)`.
- We can write a linked list 1→2-→3→4→empty as `'(1 2 3 4)`
- To get the first element in the linked list, you can write: `(first '(1 2 3 4))`↪ `1`
- To get the rest of the linked list you can write `(rest '(1 2 3 4))`↪ `'(2 3 4)`
- The empty list is represented with `'()` and you can check for the empty list with `(empty? '())` ↪ `\#t`
- We will return to discussing lists in more detail later, since they are *extremely* important.
- But for now, remember that we wanted to avoid the inconveniences given by using other existing data types to represent some piece of compound data!

The Talk

We said that we didn't want to represent all of our compound data with existing structures like lists are tuples, so let's *finally* talk about structs.

The Talk

We said that we didn't want to represent all of our compound data with existing structures like lists are tuples, so let's *finally* talk about structs.

- Let's reconsider our 2 dimensional movement program.

The Talk

We said that we didn't want to represent all of our compound data with existing structures like lists are tuples, so let's *finally* talk about structs.

- Let's reconsider our 2 dimensional movement program.
- We need a natural representation for cartesian coordiantes for the state of our world.

The Talk

We said that we didn't want to represent all of our compound data with existing structures like lists are tuples, so let's *finally* talk about structs.

- Let's reconsider our 2 dimensional movement program.
- We need a natural representation for cartesian coordiantes for the state of our world.
- We could obviously have the state of our our world be a pair `'(x . y)` or a list `'(x y)`

The Talk

We said that we didn't want to represent all of our compound data with existing structures like lists are tuples, so let's *finally* talk about structs.

- Let's reconsider our 2 dimensional movement program.
- We need a natural representation for cartesian coordiantes for the state of our world.
- We could obviously have the state of our our world be a pair `'(x . y)` or a list `'(x y)`
- But it would be better if we had piece of compound data with two fields, one field named x to represent the x coordinate and similarly a y...

The Talk

We said that we didn't want to represent all of our compound data with existing structures like lists are tuples, so let's *finally* talk about structs.

- Let's reconsider our 2 dimensional movement program.
- We need a natural representation for cartesian coordiantes for the state of our world.
- We could obviously have the state of our our world be a pair `'(x . y)` or a list `'(x y)`
- But it would be better if we had piece of compound data with two fields, one field named x to represent the x coordinate and similarly a y...
- To define the struct: `(struct point [x y])`

The Talk

We said that we didn't want to represent all of our compound data with existing structures like lists are tuples, so let's *finally* talk about structs.

- Let's reconsider our 2 dimensional movement program.
- We need a natural representation for cartesian coordiantes for the state of our world.
- We could obviously have the state of our our world be a pair `'(x . y)` or a list `'(x y)`
- But it would be better if we had piece of compound data with two fields, one field named x to represent the x coordinate and similarly a y...
- To define the struct: `(struct point [x y])`
- To make a new point: `(define one-two (point 1 2))`

The Talk

We said that we didn't want to represent all of our compound data with existing structures like lists are tuples, so let's *finally* talk about structs.

- Let's reconsider our 2 dimensional movement program.
- We need a natural representation for cartesian coordiantes for the state of our world.
- We could obviously have the state of our our world be a pair `'(x . y)` or a list `'(x y)`
- But it would be better if we had piece of compound data with two fields, one field named x to represent the x coordinate and similarly a y...
- To define the struct: `(struct point [x y])`
- To make a new point: `(define one-two (point 1 2))`
- To get the x-coordinate: `(point-x one-two)`

Writing Simple Programs With Structs

Before we consider writing more complex applications involving structs, let's consider writing a simple function.

Writing Simple Programs With Structs

Before we consider writing more complex applications involving structs, let's consider writing a simple function.

- Let's consider computing the distance to the origin where we take in one parameter that is a point, instead of taking two parameters for an x-coordinate and y-coordinate.

Writing Simple Programs With Structs

Before we consider writing more complex applications involving structs, let's consider writing a simple function.

- Let's consider computing the distance to the origin where we take in one parameter that is a point, instead of taking two parameters for an x-coordinate and y-coordinate.
- As will happen many times in this course, we will introduce the concept by examples before we cover how our design recipes change to address this new feature

Writing Simple Programs With Structs

Before we consider writing more complex applications involving structs, let's consider writing a simple function.

- Let's consider computing the distance to the origin where we take in one parameter that is a point, instead of taking two parameters for an x-coordinate and y-coordinate.
- As will happen many times in this course, we will introduce the concept by examples before we cover how our design recipes change to address this new feature

```
;;Point -> Number
;;Compute a point's distance from the origin
(define (distance-to-0 ap) 0)
```

-

Writing Simple Programs With Structs

Before we consider writing more complex applications involving structs, let's consider writing a simple function.

- Let's consider computing the distance to the origin where we take in one parameter that is a point, instead of taking two parameters for an x-coordinate and y-coordinate.
- As will happen many times in this course, we will introduce the concept by examples before we cover how our design recipes change to address this new feature

```
;;Point -> Number
;;Compute a point's distance from the origin
(define (distance-to-0 ap) 0)
```

-
- Let's add functional examples as tests:

Writing Simple Programs With Structs

Before we consider writing more complex applications involving structs, let's consider writing a simple function.

- Let's consider computing the distance to the origin where we take in one parameter that is a point, instead of taking two parameters for an x-coordinate and y-coordinate.
- As will happen many times in this course, we will introduce the concept by examples before we cover how our design recipes change to address this new feature

```
;;Point -> Number
;;Compute a point's distance from the origin
(define (distance-to-0 ap) 0)
```

-
- Let's add functional examples as tests:

```
(check-expect (distance-to-0 (point 0 5)) 5)
(check-expect (distance-to-0 (point 7 0)) 7)
```

-

Structs and Skeletons

We now can take inventory (step 4) and add a skeleton for our function. In this case we should know that we have to project out the x-coordinate and the y-coordinate in our distance function.

Structs and Skeletons

We now can take inventory (step 4) and add a skeleton for our function. In this case we should know that we have to project out the x-coordinate and the y-coordinate in our distance function.

```
(define (distance-to-0 ap)
  (... (point-x ap) ...
   ... (point-y ap) ...))
```

-

Structs and Skeletons

We now can take inventory (step 4) and add a skeleton for our function. In this case we should know that we have to project out the x-coordinate and the y-coordinate in our distance function.

```
(define (distance-to-0 ap)
  (... (point-x ap) ...
   ... (point-y ap) ...))
```

-
- To code, we essentially have the same logic as when we had a previous version of this function that took in two parameters. Now, we just use the projected out x-coordinate and y-coordinate from our point.

Structs and Skeletons

We now can take inventory (step 4) and add a skeleton for our function. In this case we should know that we have to project out the x-coordinate and the y-coordinate in our distance function.

```
(define (distance-to-0 ap)
  (... (point-x ap) ...
   ... (point-y ap) ...))
```

-
- To code, we essentially have the same logic as when we had a previous version of this function that took in two parameters. Now, we just use the projected out x-coordinate and y-coordinate from our point.

```
(define (distance-to-0 ap)
  (sqrt
    (+ (sqr (point-x ap))
       (sqr (point-y ap)))))
```

-

Structs in General

Testing that function is simple, so let's just move on to talking about structs in general.

Structs in General

Testing that function is simple, so let's just move on to talking about structs in general.

- You can define a struct in general with:
  `(struct s-name [field-name-1 ...field-name-n])`

Structs in General

Testing that function is simple, so let's just move on to talking about structs in general.

- You can define a struct in general with:
  `(struct s-name [field-name-1 ...field-name-n])`
- After creating a struct, 3 kinds of functions are automatically made for you.

Structs in General

Testing that function is simple, so let's just move on to talking about structs in general.

- You can define a struct in general with:
  `(struct s-name [field-name-1 ...field-name-n])`
- After creating a struct, 3 kinds of functions are automatically made for you.
  1. One constructor, a function that creates structure instances. It takes as many values as there are fields; as mentioned, structure is short for structure instance. The phrase structure type is a generic name for the collection of all possible instances;

Structs in General

Testing that function is simple, so let's just move on to talking about structs in general.

- You can define a struct in general with:
  `(struct s-name [field-name-1 ...field-name-n])`
- After creating a struct, 3 kinds of functions are automatically made for you.
  1. One constructor, a function that creates structure instances. It takes as many values as there are fields; as mentioned, structure is short for structure instance. The phrase structure type is a generic name for the collection of all possible instances;
  2. One selector per field, which extracts the value of the field from a structure instance; and

Structs in General

Testing that function is simple, so let's just move on to talking about structs in general.

- You can define a struct in general with:
  `(struct s-name [field-name-1 ...field-name-n])`
- After creating a struct, 3 kinds of functions are automatically made for you.
  1. One constructor, a function that creates structure instances. It takes as many values as there are fields; as mentioned, structure is short for structure instance. The phrase structure type is a generic name for the collection of all possible instances;
  2. One selector per field, which extracts the value of the field from a structure instance; and
  3. One structure predicate, which, like ordinary predicates, distinguishes instances from all other kinds of values.

Basic Struct Options

Let's illustrate each of these three kinds of functions, with our point struct.

Basic Struct Options

Let's illustrate each of these three kinds of functions, with our point struct.

1. The constructor is: `(point x-val y-val)` where x-val and y-val will be values passed
   to the x-field and y-field in our point struct. The general form is `(struct-name field-name-1-arg ... field-name-n-arg)`

Basic Struct Options

Let's illustrate each of these three kinds of functions, with our point struct.

1. The constructor is: `(point x-val y-val)` where x-val and y-val will be values passed
   to the x-field and y-field in our point struct. The general form is `(struct-name field-name-1-arg ... field-name-n-arg)`
2. The selectors per field are `(point-x point-val)` and `(point-y point-val)`. The general form of a selector for a specific field is `(struct-name-field-name val)`

Basic Struct Options

Let's illustrate each of these three kinds of functions, with our point struct.

1. The constructor is: `(point x-val y-val)` where x-val and y-val will be values passed
   to the x-field and y-field in our point struct. The general form is `(struct-name field-name-1-arg ... field-name-n-arg)`
2. The selectors per field are `(point-x point-val)` and `(point-y point-val)`. The general form of a selector for a specific field is `(struct-name-field-name val)`
3. A predicate for checking types is automatically created, for example: `(point? point-val)` and in general a predicate `struct?` is created.

Examples of Structs

Here are some basic examples of structs:

Examples of Structs

Here are some basic examples of structs:

- `(struct movie [title producer year])`

Examples of Structs

Here are some basic examples of structs:

- `(struct movie [title producer year])`
- `(struct person [name hair eyes phone])`

Examples of Structs

Here are some basic examples of structs:

- `(struct movie [title producer year])`
- `(struct person [name hair eyes phone])`
- You guys should be able to think of many more examples.

Examples of Structs

Here are some basic examples of structs:

- `(struct movie [title producer year])`
- `(struct person [name hair eyes phone])`
- You guys should be able to think of many more examples.
- **Sample Problem** Develop a structure type definition for a program that deals with bouncing balls,. The balls location is a single number, namely the distance of pixels from the top. Its constant speed is the number of pixels it moves per clock tick. Its velocity is the speed plus the direction in which it moves.

Designing Our Ball Struct

Since we are talking about a ball that bounces up and down, our structure definition is pretty simple. We need a single number for the y position and single number for the velocity in the y-axis.

Designing Our Ball Struct

Since we are talking about a ball that bounces up and down, our structure definition is pretty simple. We need a single number for the y position and single number for the velocity in the y-axis.

- `(struct ball [location vec])`

Designing Our Ball Struct

Since we are talking about a ball that bounces up and down, our structure definition is pretty simple. We need a single number for the y position and single number for the velocity in the y-axis.

- `(struct ball [location vec])`
- This is simple because our ball is moving in a single direction. But if we had a Brick Breaker esque game then we would have a bouncing ball that travels along a 2D plane, then our definition is much more complicated.

Designing Our Ball Struct

Since we are talking about a ball that bounces up and down, our structure definition is pretty simple. We need a single number for the y position and single number for the velocity in the y-axis.

- `(struct ball [location vec])`
- This is simple because our ball is moving in a single direction. But if we had a Brick Breaker esque game then we would have a bouncing ball that travels along a 2D plane, then our definition is much more complicated.
- Let's first consider defining a 2D vector struct as follows: `(struct vector [delta-x delta-y])`

Designing Our Ball Struct

Since we are talking about a ball that bounces up and down, our structure definition is pretty simple. We need a single number for the y position and single number for the velocity in the y-axis.

- `(struct ball [location vec])`
- This is simple because our ball is moving in a single direction. But if we had a Brick Breaker esque game then we would have a bouncing ball that travels along a 2D plane, then our definition is much more complicated.
- Let's first consider defining a 2D vector struct as follows: `(struct vector [delta-x delta-y])`
- Now, we can represent a ball as a point (which only has positive components) and a vector (which can have negative components): `(struct 2D-ball position vec)`

Other Representations

Our 2D Ball struct has nested occurrences of other structs. This is a natural thing, and even recursive descriptions of data are natural, i.e. linked lists and binary trees. But we can also consider using a *flat representation* for our 2D Ball, which doesn't nest structs.

Other Representations

Our 2D Ball struct has nested occurrences of other structs. This is a natural thing, and even recursive descriptions of data are natural, i.e. linked lists and binary trees. But we can also consider using a *flat representation* for our 2D Ball, which doesn't nest structs.

- `(struct 2D-ball [x y delta-x delta-y]`

Other Representations

Our 2D Ball struct has nested occurrences of other structs. This is a natural thing, and even recursive descriptions of data are natural, i.e. linked lists and binary trees. But we can also consider using a *flat representation* for our 2D Ball, which doesn't nest structs.

- `(struct 2D-ball [x y delta-x delta-y]`
- Although valid, I think it's better to keep representations natural and just nest things, barring performance concerns.

Other Representations

Our 2D Ball struct has nested occurrences of other structs. This is a natural thing, and even recursive descriptions of data are natural, i.e. linked lists and binary trees. But we can also consider using a *flat representation* for our 2D Ball, which doesn't nest structs.

- `(struct 2D-ball [x y delta-x delta-y]`
- Although valid, I think it's better to keep representations natural and just nest things, barring performance concerns.
- Let's talk about defining data definitions for structs. We must specify the form of the struct and the types of its field and provide an interpretation of what each of the fields represents. Here's how we do this for our point struct:

Other Representations

Our 2D Ball struct has nested occurrences of other structs. This is a natural thing, and even recursive descriptions of data are natural, i.e. linked lists and binary trees. But we can also consider using a *flat representation* for our 2D Ball, which doesn't nest structs.

- `(struct 2D-ball [x y delta-x delta-y]`
- Although valid, I think it's better to keep representations natural and just nest things, barring performance concerns.
- Let's talk about defining data definitions for structs. We must specify the form of the struct and the types of its field and provide an interpretation of what each of the fields represents. Here's how we do this for our point struct:

```
(define-struct point [x y])
; A Point is a structure:
;    (point Number Number)
; interpretation a point x pixels from left, y from top
```

-

Computing With Structures

As mentioned, we could have use tuples instead of structs to represent compound data, but remembering to access the name field of a person struct is a lot easier than remembering to project out the 7th element in some n-tuple (assuming n¿=7).

Computing With Structures

As mentioned, we could have use tuples instead of structs to represent compound data, but remembering to access the name field of a person struct is a lot easier than remembering to project out the 7th element in some n-tuple (assuming n¿=7).

- How does *computing* with structures become more natural than simply computing with tuples?

Computing With Structures

As mentioned, we could have use tuples instead of structs to represent compound data, but remembering to access the name field of a person struct is a lot easier than remembering to project out the 7th element in some n-tuple (assuming n¿=7).

- How does *computing* with structures become more natural than simply computing with tuples?
- We can provide a natural data representation that simplifies the coding process by giving us easy to remember field-names to select.

Computing With Structures

As mentioned, we could have use tuples instead of structs to represent compound data, but remembering to access the name field of a person struct is a lot easier than remembering to project out the 7th element in some n-tuple (assuming n¿=7).

- How does *computing* with structures become more natural than simply computing with tuples?
- We can provide a natural data representation that simplifies the coding process by giving us easy to remember field-names to select.
- Let us consider how to relate a struct description to a diagram that illustrates its "structure".

Computing With Structures

As mentioned, we could have use tuples instead of structs to represent compound data, but remembering to access the name field of a person struct is a lot easier than remembering to project out the 7th element in some n-tuple (assuming n¿=7).

- How does *computing* with structures become more natural than simply computing with tuples?
- We can provide a natural data representation that simplifies the coding process by giving us easy to remember field-names to select.
- Let us consider how to relate a struct description to a diagram that illustrates its "structure".
- `(struct centry [name home office cell])`

Struct Structure

So, for cell phone entry structs we had three fields that can contain possible values. Consider the following concrete one:

Struct Structure

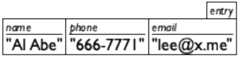
So, for cell phone entry structs we had three fields that can contain possible values. Consider the following concrete one:

- `(define pl (centry "Al Abe" "666-7771" "lee@x.me"))`

Struct Structure

So, for cell phone entry structs we had three fields that can contain possible values. Consider the following concrete one:

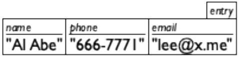
- `(define pl (centry "Al Abe" "666-7771" "lee@x.me"))`
- This has the following visual representation:



Struct Structure

So, for cell phone entry structs we had three fields that can contain possible values. Consider the following concrete one:

- `(define pl (centry "Al Abe" "666-7771" "lee@x.me"))`
- This has the following visual representation:

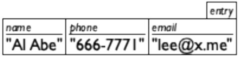


- In a sense, calling `(centry-name pl)` is unlocking a box in the struct, with a specific key that allows you to retrieve the underlying value. In general we can think of field access as a kind of “unboxing”.

Struct Structure

So, for cell phone entry structs we had three fields that can contain possible values. Consider the following concrete one:

- `(define pl (centry "Al Abe" "666-7771" "lee@x.me"))`
- This has the following visual representation:



- In a sense, calling `(centry-name pl)` is unlocking a box in the struct, with a specific key that allows you to retrieve the underlying value. In general we can think of field access as a kind of "unboxing".
- Using a "key" to unlock the wrong box raises a runtime error `(centry-name (point 1 2))` $\hookrightarrow$ entry-name:expects a centry, given (point 42 5)

Interpreting Structure

In general, we need to think about how we describe the interpretation of structs.

Interpreting Structure

In general, we need to think about how we describe the interpretation of structs.

- Defining a data interpretation for the ball with one dimensional movement is simple. We can describe position and the velocity in terms of numbers.

Interpreting Structure

In general, we need to think about how we describe the interpretation of structs.

- Defining a data interpretation for the ball with one dimensional movement is simple. We can describe position and the velocity in terms of numbers.

  ```
  (define-struct ball [location velocity])
  ; A Ball-1d is a structure:
  ;   (ball Number Number)
  ; interpretation 1 distance to top and velocity
  ; interpretation 2 distance to left and velocity
  ```

-

Interpreting Structure

In general, we need to think about how we describe the interpretation of structs.

- Defining a data interpretation for the ball with one dimensional movement is simple. We can describe position and the velocity in terms of numbers.

```
(define-struct ball [location velocity])
; A Ball-1d is a structure:
;    (ball Number Number)
; interpretation 1 distance to top and velocity
; interpretation 2 distance to left and velocity
```

-
- Interestingly, we can give 2 interpretations, depending on if the ball is moving vertically (interpretation 1) or horizontally (interpretation 2)

Interpreting Structure (cont.)

Interpretations become slightly more interesting with nested structures.

Interpreting Structure (cont.)

Interpretations become slightly more interesting with nested structures.

- Consider the ball that can move in two dimensions.

Interpreting Structure (cont.)

Interpretations become slightly more interesting with nested structures.

- Consider the ball that can move in two dimensions.
- Since the ball relies on a vector struct, it is important to make sure that the vector also has an interpretation

Interpreting Structure (cont.)

Interpretations become slightly more interesting with nested structures.

- Consider the ball that can move in two dimensions.
- Since the ball relies on a vector struct, it is important to make sure that the vector also has an interpretation

```
; A Ball-2d is a structure:
;   (ball Point Vel)
; interpretation a 2-dimensional position and velocity

(define-struct vel [deltax deltay])
; A Vel is a structure:
;   (vel Number Number)
; interpretation (make-vel dx dy) means a velocity of
; dx pixels [per tick] along the horizontal and
; dy pixels [per tick] along the vertical direction
```

-

Interpreting Structure (cont.)

Recursively defined types don't add any problems to writing data interpretations.

Interpreting Structure (cont.)

Recursively defined types don't add any problems to writing data interpretations.

- What are some famous recursively defined types?

Interpreting Structure (cont.)

Recursively defined types don't add any problems to writing data interpretations.

- What are some famous recursively defined types?
- Let's consider a linked list.

Interpreting Structure (cont.)

Recursively defined types don't add any problems to writing data interpretations.

- What are some famous recursively defined types?
- Let's consider a linked list.

```
; A LinkedList is a structure:
; (linked-list T LinkedList)
; interpretation a value contained in the current node
; and a reference to the rest of the list
```

-

Interpreting Structure (cont.)

Recursively defined types don't add any problems to writing data interpretations.

- What are some famous recursively defined types?
- Let's consider a linked list.

```
; A LinkedList is a structure:
; (linked-list T LinkedList)
; interpretation a value contained in the current node
; and a reference to the rest of the list
```

-
- We don't have to worry about the recursive instance of the type, since we are currently interpreting it.

Interpreting Structure (cont.)

Recursively defined types don't add any problems to writing data interpretations.

- What are some famous recursively defined types?
- Let's consider a linked list.

```
; A LinkedList is a structure:
; (linked-list T LinkedList)
; interpretation a value contained in the current node
; and a reference to the rest of the list
```

-
- We don't have to worry about the recursive instance of the type, since we are currently interpreting it.
- The T just communicates that we are defining linked lists that work over any type (*parametric polymorphism* otherwise known as *generics* in Java)

Structs and Program Design

Now we need to consider designing programs using structs.
Sample Problem Your team is designing an interactive game program that moves a red dot across a image canvas and allows players to use the mouse to reset the dot. Here is how far you got together:

Structs and Program Design

Now we need to consider designing programs using structs.

Sample Problem Your team is designing an interactive game program that moves a red dot across a image canvas and allows players to use the mouse to reset the dot. Here is how far you got together:

```
(define MTS (empty-scene 100 100))
(define DOT (circle 3 "solid" "red"))

; A Point represents the state of the world.

; Point -> Point
(define (main p0)
  (big-bang p0
    [on-tick x+]
    [on-mouse reset-dot]
    [to-draw scene+dot]))
```

Designing scene+dot

Let us first assume that you're tasked with designing `scene+dot`.

Designing scene+dot

Let us first assume that you're tasked with designing `scene+dot`.

- Of course, we already have our data interpretation so we start with step 2 and provide our signature, statement of purpose, and function header.

Designing scene+dot

Let us first assume that you're tasked with designing `scene+dot`.

- Of course, we already have our data interpretation so we start with step 2 and provide our signature, statement of purpose, and function header.

```
; Point -> Image
; adds a red spot to MTS at p
(define (scene+dot p) MTS)
```

-

Designing scene+dot

Let us first assume that you're tasked with designing `scene+dot`.

- Of course, we already have our data interpretation so we start with step 2 and provide our signature, statement of purpose, and function header.

```
; Point -> Image
; adds a red spot to MTS at p
(define (scene+dot p) MTS)
```

-
- Finishing step 3 and creating the following functional examples (as tests) is straightforward:

Designing scene+dot

Let us first assume that you're tasked with designing `scene+dot`.

- Of course, we already have our data interpretation so we start with step 2 and provide our signature, statement of purpose, and function header.

```
; Point -> Image
; adds a red spot to MTS at p
(define (scene+dot p) MTS)
```

-

- Finishing step 3 and creating the following functional examples (as tests) is straightforward:

```
(check-expect (scene+dot (point 10 20))
              (place-image DOT 10 20 MTS))
(check-expect (scene+dot (point 88 73))
              (place-image DOT 88 73 MTS))
```

-

Designing scene+dot (cont.)

We can now move on to carrying out step 4 and taking inventory:

Designing scene+dot (cont.)

We can now move on to carrying out step 4 and taking inventory:

```
(define (scene+dot p)
  (... (point-x p) ... (point-y p) ...))
```

-

Designing scene+dot (cont.)

We can now move on to carrying out step 4 and taking inventory:

```
(define (scene+dot p)
  (... (point-x p) ... (point-y p) ...))
```

-
- Finishing step 5 is easy, we simply need to place the projected x and y coordinates as arguments to `place-image`

Designing scene+dot (cont.)

We can now move on to carrying out step 4 and taking inventory:

```
(define (scene+dot p)
  (... (point-x p) ... (point-y p) ...))
```

-
- Finishing step 5 is easy, we simply need to place the projected x and y coordinates as arguments to `place-image`

```
(define (scene+dot p)
  (place-image DOT (point-x p) (point-y p) MTS))
```

-

Designing scene+dot (cont.)

We can now move on to carrying out step 4 and taking inventory:

```
(define (scene+dot p)
  (... (point-x p) ... (point-y p) ...))
```

-
- Finishing step 5 is easy, we simply need to place the projected x and y coordinates as arguments to `place-image`

```
(define (scene+dot p)
  (place-image DOT (point-x p) (point-y p) MTS))
```

-
- Testing this is uninteresting, so let's consider if we were asked to define the `x+` function, which takes in a Point and returns a new Point with an x-coordinate that is 3 units further to the right of the old point.

Designing x+

Designing our `x+` function is a bit harder than `scene+dot`, because we are producing structures as output.

Designing x+

Designing our `x+` function is a bit harder than `scene+dot`, because we are producing structures as output.

- When carrying out step 2, recall that `x+` handles clock ticks:

```
;;Point -> Point
;;Updates the position of our dot at each clock tick
(define (x+ p) p)
```

Designing x+

Designing our `x+` function is a bit harder than `scene+dot`, because we are producing structures as output.

- When carrying out step 2, recall that `x+` handles clock ticks:

```
;;Point -> Point
;;Updates the position of our dot at each clock tick
(define (x+ p) p)
```

- Coming up with functional examples as tests is straightforward:

```
(check-expect (x+ (point 0 0)) (point 3 0))
(check-expect (x+ (point 10 10)) (point 13 10))
(check-expect (x+ (point 0 10)) (point 3 10))
```

Designing x+

Designing our `x+` function is a bit harder than `scene+dot`, because we are producing structures as output.

- When carrying out step 2, recall that `x+` handles clock ticks:

```
;;Point -> Point
;;Updates the position of our dot at each clock tick
(define (x+ p) p)
```

- Coming up with functional examples as tests is straightforward:

```
(check-expect (x+ (point 0 0)) (point 3 0))
(check-expect (x+ (point 10 10)) (point 13 10))
(check-expect (x+ (point 0 10)) (point 3 10))
```

- To take inventory we project out the x and y fields from our point, as usual:

```
(define (x+ p) (... (point-x p) ... (point-y p) ...)
```

Designing x+ (cont.)

To finish coding, we need to first add 3 to the x-coordinate and then pack the result back into a new point structure.

Designing x+ (cont.)

To finish coding, we need to first add 3 to the x-coordinate and then pack the result back into a new point structure.

```
(define (x+ p)
  (point (+ (point-x p) 3) (point-y p)))
```

-

Designing x+ (cont.)

To finish coding, we need to first add 3 to the x-coordinate and then pack the result back into a new point structure.

```
(define (x+ p)
  (point (+ (point-x p) 3) (point-y p)))
```

-
- This is a perfect example of the *essence* of functional programming. We are creating a *new* point whose x-coordinate is based on the old one, instead of modifying the original point to store a new x-coordinate.

Designing x+ (cont.)

To finish coding, we need to first add 3 to the x-coordinate and then pack the result back into a new point structure.

```
(define (x+ p)
  (point (+ (point-x p) 3) (point-y p)))
```

-
- This is a perfect example of the *essence* of functional programming. We are creating a *new* point whose x-coordinate is based on the old one, instead of modifying the original point to store a new x-coordinate.
- But there is something inelegant about this, right?

Designing x+ (cont.)

To finish coding, we need to first add 3 to the x-coordinate and then pack the result back into a new point structure.

```
(define (x+ p)
  (point (+ (point-x p) 3) (point-y p)))
```

-
- This is a perfect example of the *essence* of functional programming. We are creating a *new* point whose x-coordinate is based on the old one, instead of modifying the original point to store a new x-coordinate.
- But there is something inelegant about this, right?
- If we had more fields than y, we simply are projecting out the old field as an argument when creating the new struct value.

Designing x+ (cont.)

To finish coding, we need to first add 3 to the x-coordinate and then pack the result back into a new point structure.

```
(define (x+ p)
  (point (+ (point-x p) 3) (point-y p)))
```

-
- This is a perfect example of the *essence* of functional programming. We are creating a *new* point whose x-coordinate is based on the old one, instead of modifying the original point to store a new x-coordinate.
- But there is something inelegant about this, right?
- If we had more fields than y, we simply are projecting out the old field as an argument when creating the new struct value.
- This adds a lot of boilerplate code...

Struct Boilerplate

We might want to define a function, `point-set-x` which takes in a point and a value and produces a new point where the x-coordinate is the given value and the y-coordinate is taken from the old point.

Struct Boilerplate

We might want to define a function, `point-set-x` which takes in a point and a value and produces a new point where the x-coordinate is the given value and the y-coordinate is taken from the old point.

- Here is the code:

```
(define (point-set-x p x)
  (point x (point-y p)))
```

Struct Boilerplate

We might want to define a function, `point-set-x` which takes in a point and a value and produces a new point where the x-coordinate is the given value and the y-coordinate is taken from the old point.

- Here is the code:

```
(define (point-set-x p x)
  (point x (point-y p)))
```

- We can now redefine `x+` with it:

```
(define (x+ p)
  (point-set-x p (+ (point-x p) 3)))
```

Struct Boilerplate

We might want to define a function, `point-set-x` which takes in a point and a value and produces a new point where the x-coordinate is the given value and the y-coordinate is taken from the old point.

- Here is the code:
  ```
  (define (point-set-x p x)
    (point x (point-y p)))
  ```
- We can now redefine `x+` with it:
  ```
  (define (x+ p)
    (point-set-x p (+ (point-x p) 3)))
  ```
- `point-set-x` is known as a *functional setter*, similar to a more traditional setter in languages like Java.

Struct Boilerplate

We might want to define a function, `point-set-x` which takes in a point and a value and produces a new point where the x-coordinate is the given value and the y-coordinate is taken from the old point.

- Here is the code:
```
(define (point-set-x p x)
  (point x (point-y p)))
```
- We can now redefine `x+` with it:
```
(define (x+ p)
  (point-set-x p (+ (point-x p) 3)))
```
- `point-set-x` is known as a *functional setter*, similar to a more traditional setter in languages like Java.
- However, defining an update operation on a complicated structure can get very complicated, and we can get around this uses *lenses* (we may discuss this later in the course).

Events Creating Structs

We can consider designing a function `reset-dot` which places a dot where a mouse is clicked.

Events Creating Structs

We can consider designing a function `reset-dot` which places a dot where a mouse is clicked.

- We start with step 2 and create a signature, statement of purpose, and stub for this function.

Events Creating Structs

We can consider designing a function `reset-dot` which places a dot where a mouse is clicked.

- We start with step 2 and create a signature, statement of purpose, and stub for this function.

  ```
  ; Point Number Number MouseEvt -> Point
  ; for mouse clicks, (point x y); otherwise p
  (define (reset-dot p x y me) p)
  ```

-

Events Creating Structs

We can consider designing a function `reset-dot` which places a dot where a mouse is clicked.

- We start with step 2 and create a signature, statement of purpose, and stub for this function.

```
; Point Number Number MouseEvt -> Point
; for mouse clicks, (point x y); otherwise p
(define (reset-dot p x y me) p)
```

-
- The statement of purpose and the signature should make designing the functional examples easy.

Events Creating Structs

We can consider designing a function `reset-dot` which places a dot where a mouse is clicked.

- We start with step 2 and create a signature, statement of purpose, and stub for this function.
  ```
  ; Point Number Number MouseEvt -> Point
  ; for mouse clicks, (point x y); otherwise p
  (define (reset-dot p x y me) p)
  ```
-
- The statement of purpose and the signature should make designing the functional examples easy.
  ```
  (check-expect
    (reset-dot (point 10 20) 29 31 "button-down")
    (point 29 31))
  (check-expect
    (reset-dot (point 10 20) 29 31 "button-up")
    (point 10 20))
  ```
-

Events Creating Structs (cont.)

From here, writing the skeleton and finishing coding are easy.

Events Creating Structs (cont.)

From here, writing the skeleton and finishing coding are easy.

- The Skeleton: 

Events Creating Structs (cont.)

From here, writing the skeleton and finishing coding are easy.

- The Skeleton: 
- For real this time:

```
(define (reset-dot p x y me)
  (cond
    [(mouse=? "button-down" me) (... p ... x y ...)]
    [else (... p ... x y ...)]))
```

Events Creating Structs (cont.)

Finally, though the skeleton looks complicated the final version of the function is relatively simple.

Events Creating Structs (cont.)

Finally, though the skeleton looks complicated the final version of the function is relatively simple.

```
(define (reset-dot p x y me)
  (cond
    [(mouse=? me "button-down") (point x y)]
    [else p]))
```

•

Events Creating Structs (cont.)

Finally, though the skeleton looks complicated the final version of the function is relatively simple.

```
(define (reset-dot p x y me)
  (cond
    [(mouse=? me "button-down") (point x y)]
    [else p]))
```

-
- The takeaway here is that our skeletons can end up being more complicated than the actual final version.

Events Creating Structs (cont.)

Finally, though the skeleton looks complicated the final version of the function is relatively simple.

```
(define (reset-dot p x y me)
  (cond
    [(mouse=? me "button-down") (point x y)]
    [else p]))
```

-
- The takeaway here is that our skeletons can end up being more complicated than the actual final version.
- This is especially true when we consider skeletoning out code for which a parameter is a struct.

Complex Skeletons

Consider a skeleton for a function whose input is a ball moving two dimensionally.

Complex Skeletons

Consider a skeleton for a function whose input is a ball moving two dimensionally.

- We would have to write a field access for both of the underlying point structure's fields, along with field accesses for the underlying vector structure's fields.

Complex Skeletons

Consider a skeleton for a function whose input is a ball moving two dimensionally.

- We would have to write a field access for both of the underlying point structure's fields, along with field accesses for the underlying vector structure's fields.
- It would look like the following:

```
(define (complex-skeleton ball)
  (... (point-x (ball-position ball)) ...
   ... (point-y (ball-position ball)) ...
   ... (vec-delta-x (ball-vector ball)) ...
   ... (vec-delta-y (ball-vector ball)) ...))
```

Complex Skeletons

Consider a skeleton for a function whose input is a ball moving two dimensionally.

- We would have to write a field access for both of the underlying point structure's fields, along with field accesses for the underlying vector structure's fields.
- It would look like the following:

```
(define (complex-skeleton ball)
  (... (point-x (ball-position ball)) ...
   ... (point-y (ball-position ball)) ...
   ... (vec-delta-x (ball-vector ball)) ...
   ... (vec-delta-y (ball-vector ball)) ...))
```

- For some of the classes you see in Java, skeletoning out code in such a manner is infeasible...

So, What's a Feasible Way?

So, how should we skeleton out functions with complex structs as input?

So, What's a Feasible Way?

So, how should we skeleton out functions with complex structs as input?

- **If a function deals with nested structures, develop one function per level of nesting.**

So, What's a Feasible Way?

So, how should we skeleton out functions with complex structs as input?

- **If a function deals with nested structures, develop one function per level of nesting.**
- For our previous example, we could have wrote:

```
(define (complex-skeleton ball)
  (... (point-x (ball-position ball)) ...
   ... (vec-delta-y (ball-vector ball)) ...))
```

So, What's a Feasible Way?

So, how should we skeleton out functions with complex structs as input?

- **If a function deals with nested structures, develop one function per level of nesting.**
- For our previous example, we could have wrote:

```
(define (complex-skeleton ball)
  (... (point-x (ball-position ball)) ...
   ... (vec-delta-y (ball-vector ball)) ...))
```

- Of course, things are easier if we picked the write functions to add in our skeleton, but this keeps the structure of the skeleton much simpler.

So, What's a Feasible Way?

So, how should we skeleton out functions with complex structs as input?

- **If a function deals with nested structures, develop one function per level of nesting.**
- For our previous example, we could have wrote:

```
(define (complex-skeleton ball)
  (... (point-x (ball-position ball)) ...
   ... (vec-delta-y (ball-vector ball)) ...))
```

- Of course, things are easier if we picked the write functions to add in our skeleton, but this keeps the structure of the skeleton much simpler.
- We will return to this point later in the course.

A Digression About Signatures

Signatures can tell us a lot about a function. I'll put some signatures up, and what kind of functions could we write for these signatures?

- Number Number $\rightarrow$ Number

A Digression About Signatures

Signatures can tell us a lot about a function. I'll put some signatures up, and what kind of functions could we write for these signatures?

- Number Number $\rightarrow$ Number
- Point Vec $\rightarrow$ Point

A Digression About Signatures

Signatures can tell us a lot about a function. I'll put some signatures up, and what kind of functions could we write for these signatures?

- Number Number $\rightarrow$ Number
- Point Vec $\rightarrow$ Point
- Point $\rightarrow$ Number

A Digression About Signatures

Signatures can tell us a lot about a function. I'll put some signatures up, and what kind of functions could we write for these signatures?

- Number Number $\rightarrow$ Number
- Point Vec $\rightarrow$ Point
- Point $\rightarrow$ Number
- Vec $\rightarrow$ Number

A Digression About Signatures

Signatures can tell us a lot about a function. I'll put some signatures up, and what kind of functions could we write for these signatures?

- Number Number $\rightarrow$ Number
- Point Vec $\rightarrow$ Point
- Point $\rightarrow$ Number
- Vec $\rightarrow$ Number
- T LinkedList $\rightarrow$ LinkedList

A Digression About Signatures

Signatures can tell us a lot about a function. I'll put some signatures up, and what kind of functions could we write for these signatures?

- Number Number $\rightarrow$ Number
- Point Vec $\rightarrow$ Point
- Point $\rightarrow$ Number
- Vec $\rightarrow$ Number
- T LinkedList $\rightarrow$ LinkedList
- T $\rightarrow$ T

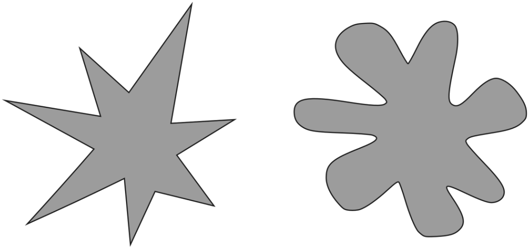
A Digression About Signatures

Signatures can tell us a lot about a function. I'll put some signatures up, and what kind of functions could we write for these signatures?

- Number Number $\rightarrow$ Number
- Point Vec $\rightarrow$ Point
- Point $\rightarrow$ Number
- Vec $\rightarrow$ Number
- T LinkedList $\rightarrow$ LinkedList
- T $\rightarrow$ T
- The idea that a function signature can tell you about the behavior of the function is an extremely powerful idea.

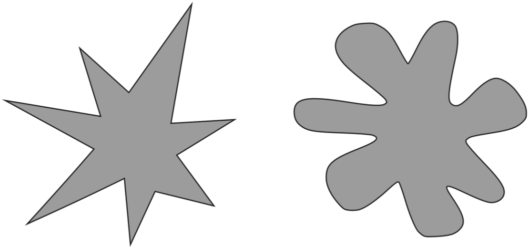
Data and the Universe

Here's a fun little experiment. Look at these two shapes:



Data and the Universe

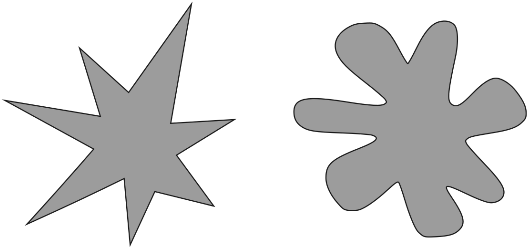
Here's a fun little experiment. Look at these two shapes:



- Which of these shapes is named Bouba and which Kiki?

Data and the Universe

Here's a fun little experiment. Look at these two shapes:



- Which of these shapes is named Bouba and which Kiki?
- This is likely due to some physical attributes of sound, but our strong preference for naming the round one Bouba and the sharp one Kiki shows that humans have **innate preferences** about the *naming* and *representation of things*.

Data and the Universe

Racket chose certain extremely powerful data types as primitives that make up our initial *data universe*.

Data and the Universe

Racket chose certain extremely powerful data types as primitives that make up our initial *data universe*.

- From here, we talked about enumerations, intervals, and itemizations as a way to restrict the kind of values from the universe that we want our function to consider.

Data and the Universe

Racket chose certain extremely powerful data types as primitives that make up our initial *data universe*.

- From here, we talked about enumerations, intervals, and itemizations as a way to restrict the kind of values from the universe that we want our function to consider.
- For example, look at this data definition:

```
; A BS is one of:
;   "hello",
;   "world", or
;   pi.
```

Data and the Universe

Racket chose certain extremely powerful data types as primitives that make up our initial *data universe*.

- From here, we talked about enumerations, intervals, and itemizations as a way to restrict the kind of values from the universe that we want our function to consider.
- For example, look at this data definition:

```
; A BS is one of:
;   "hello",
;   "world", or
;   pi.
```

- If this is the domain of some function, we restricted ourselves to needing to handle three values.

Expanding the Universe

With itemizations we specified subsets of the existing data universe.

- 

Expanding the Universe

With itemizations we specified subsets of the existing data universe.

- 

- When we added structs, we actually provided the ability to extend the universe of data.

Expanding the Universe

With itemizations we specified subsets of the existing data universe.

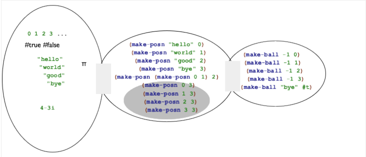
- 

- When we added structs, we actually provided the ability to extend the universe of data.
- We provided new data types describing points, balls, etc.

Expanding the Universe

With itemizations we specified subsets of the existing data universe.

- 

- When we added structs, we actually provided the ability to extend the universe of data.
- We provided new data types describing points, balls, etc.
- 

Respect Interpretations

Notice that we specify our point struct as containing two numbers.

Respect Interpretations

Notice that we specify our point struct as containing two numbers.

- But when we added this struct, we really created a pair that could contain *any* two values.

Respect Interpretations

Notice that we specify our point struct as containing two numbers.

- But when we added this struct, we really created a pair that could contain *any* two values.
- Racket doesn't inherently stop us from writing: `(point "foo" "bar")`.

Respect Interpretations

Notice that we specify our point struct as containing two numbers.

- But when we added this struct, we really created a pair that could contain *any* two values.
- Racket doesn't inherently stop us from writing: `(point "foo" "bar")`.
- Typed Racket can add the exact kind of point to the universe we want and stop invalid data from being added to a point's field at compile time.

Respect Interpretations

Notice that we specify our point struct as containing two numbers.

- But when we added this struct, we really created a pair that could contain *any* two values.
- Racket doesn't inherently stop us from writing: `(point "foo" "bar")`.
- Typed Racket can add the exact kind of point to the universe we want and stop invalid data from being added to a point's field at compile time.
- Without that, it is up to us programmers to validate that points are only constructed with valid arguments.

Respect Interpretations

Notice that we specify our point struct as containing two numbers.

- But when we added this struct, we really created a pair that could contain *any* two values.
- Racket doesn't inherently stop us from writing: `(point "foo" "bar")`.
- Typed Racket can add the exact kind of point to the universe we want and stop invalid data from being added to a point's field at compile time.
- Without that, it is up to us programmers to validate that points are only constructed with valid arguments.
- This might mean creating a function `(define (make-point x y) ...)` that checks that both `x` and `y` actually receive integers before calling the `point` constructor. More on this later.

Respect Interpretations (cont.)

Data definitions are a critical part of programming.

Respect Interpretations (cont.)

Data definitions are a critical part of programming.

- In Java when you define a class, you are saying that this class is an example of data that was important enough to need modeling. Making a data definition specifies that this kind of data is integral to understanding the program.

Respect Interpretations (cont.)

Data definitions are a critical part of programming.

- In Java when you define a class, you are saying that this class is an example of data that was important enough to need modeling. Making a data definition specifies that this kind of data is integral to understanding the program.

Respect Interpretations (cont.)

Data definitions are a critical part of programming.

- In Java when you define a class, you are saying that this class is an example of data that was important enough to need modeling. Making a data definition specifies that this kind of data is integral to understanding the program.
- So, when adding a definition provide an example of realistic data for what is being defined:

Respect Interpretations (cont.)

Data definitions are a critical part of programming.

- In Java when you define a class, you are saying that this class is an example of data that was important enough to need modeling. Making a data definition specifies that this kind of data is integral to understanding the program.
- So, when adding a definition provide an example of realistic data for what is being defined:
- For a built-in collection of data (number, string, Boolean, images), choose your favorite examples. (E.G. 1)

Respect Interpretations (cont.)

Data definitions are a critical part of programming.

- In Java when you define a class, you are saying that this class is an example of data that was important enough to need modeling. Making a data definition specifies that this kind of data is integral to understanding the program.
- So, when adding a definition provide an example of realistic data for what is being defined:
- For a built-in collection of data (number, string, Boolean, images), choose your favorite examples. (E.G. 1)
- For an enumeration, use several of the items of the enumeration. (For NorF use #f)

Respect Interpretations (cont.)

Data definitions are a critical part of programming.

- In Java when you define a class, you are saying that this class is an example of data that was important enough to need modeling. Making a data definition specifies that this kind of data is integral to understanding the program.
- So, when adding a definition provide an example of realistic data for what is being defined:
- For a built-in collection of data (number, string, Boolean, images), choose your favorite examples. (E.G. 1)
- For an enumeration, use several of the items of the enumeration. (For NorF use #f)
- For intervals, use the end points (if they are included) and at least one interior point.

Respect Interpretations (cont.)

Data definitions are a critical part of programming.

- In Java when you define a class, you are saying that this class is an example of data that was important enough to need modeling. Making a data definition specifies that this kind of data is integral to understanding the program.
- So, when adding a definition provide an example of realistic data for what is being defined:
- For a built-in collection of data (number, string, Boolean, images), choose your favorite examples. (E.G. 1)
- For an enumeration, use several of the items of the enumeration. (For NorF use #f)
- For intervals, use the end points (if they are included) and at least one interior point.
- For itemizations, deal with each part separately

Respect Interpretations (cont.)

Data definitions are a critical part of programming.

- In Java when you define a class, you are saying that this class is an example of data that was important enough to need modeling. Making a data definition specifies that this kind of data is integral to understanding the program.
- So, when adding a definition provide an example of realistic data for what is being defined:
- For a built-in collection of data (number, string, Boolean, images), choose your favorite examples. (E.G. 1)
- For an enumeration, use several of the items of the enumeration. (For NorF use #f)
- For intervals, use the end points (if they are included) and at least one interior point.
- For itemizations, deal with each part separately
- For data definitions for structures, follow the natural language description; that is, use the constructor and pick an example from the data collection named for each field.