Structures and All That

October 8, 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))`↪ `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.

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] \#:transparent)`

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] \#:transparent)`
- 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] \#:transparent)`
- 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] \#:transp`

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] \#:transp`
- 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] \#:transp`
- 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] \#:transp`
- 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] \#:transp`
- 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] \#:transparent)
  ```
  ```

Examples of Structs

Here are some basic examples of structs:

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

Examples of Structs

Here are some basic examples of structs:

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

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] \#:transparent)`
- 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] \#:transparent)`
- 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] \#:transparent)`

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] \#:transparent)`
- 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] \#:transparent)`
- 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] \#:transparent)`

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] \#:transparent)
  ```
  ```

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] \#:transparent)
  ```
- 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] \#:transparent)
  ```
- 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] \#:transparent)
  ```
- 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:

  ```
  (struct point [x y] \#:transparent)
  ; 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] \#:transparent)
  ```
  ```

Struct Structure

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

Struct Structure

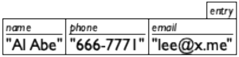
So, for cell phone entry structs we had three fields that can contain possible values. Consider the following concrete one:

- `(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:

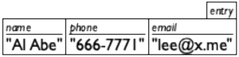
- `(define pl (centry "Al Abe" "666-7771" "lee@x.me"))`
- This has the following visual representation:



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:

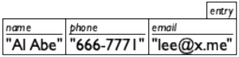


- In a sense, calling `(centry-name pl)` is unlocking a box in the struct, with a specific key that allows you to retrieve the underlying value. In general we can think of field access as a kind of "unboxing".

Struct Structure

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

- `(define p1 (centry "Al Abe" "666-7771" "lee@x.me"))`
- This has the following visual representation:



- In a sense, calling `(centry-name p1)` 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)

Some Definitions

- **Syntax** - the arrangement of words and phrases to create well-formed sentences in a language.

Some Definitions

- **Syntax** - the arrangement of words and phrases to create well-formed sentences in a language.
- **Semantics** - the branch of linguistics and logic concerned with meaning. There are a number of branches and subbranches of semantics, including formal semantics, which studies the logical aspects of meaning, such as sense, reference, implication, and logical form, lexical semantics, which studies word meanings and word relations, and conceptual semantics, which studies the cognitive structure of meaning.

Some Definitions

- **Syntax** - the arrangement of words and phrases to create well-formed sentences in a language.
- **Semantics** - the branch of linguistics and logic concerned with meaning. There are a number of branches and subbranches of semantics, including formal semantics, which studies the logical aspects of meaning, such as sense, reference, implication, and logical form, lexical semantics, which studies word meanings and word relations, and conceptual semantics, which studies the cognitive structure of meaning.
- **Interpretation** - the action of explaining the meaning of something.

Some Definitions

- **Syntax** - the arrangement of words and phrases to create well-formed sentences in a language.
- **Semantics** - the branch of linguistics and logic concerned with meaning. There are a number of branches and subbranches of semantics, including formal semantics, which studies the logical aspects of meaning, such as sense, reference, implication, and logical form, lexical semantics, which studies word meanings and word relations, and conceptual semantics, which studies the cognitive structure of meaning.
- **Interpretation** - the action of explaining the meaning of something.
- **Denotation** - the object or concept to which a term refers, or the set of objects of which a predicate is true.

A Few More

- **Compiler** - a computer program that translates computer code written in one programming language (the source language) into another language (the target language).

A Few More

- **Compiler** - a computer program that translates computer code written in one programming language (the source language) into another language (the target language).
- **Computable Function** - A function that can be represented by a *general recursive function*, a **lambda computable function**, or a **Turing Machine**

A Few More

- **Compiler** - a computer program that translates computer code written in one programming language (the source language) into another language (the target language).
- **Computable Function** - A function that can be represented by a *general recursive function*, a **lambda computable function**, or a **Turing Machine**
- **Programming Language** - a formal language, which comprises a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms.

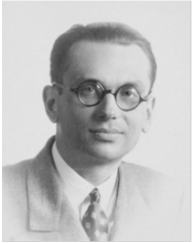
A Few More

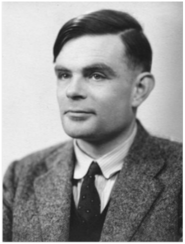
- **Compiler** - a computer program that translates computer code written in one programming language (the source language) into another language (the target language).
- **Computable Function** - A function that can be represented by a *general recursive function*, a **lambda computable function**, or a **Turing Machine**
- **Programming Language** - a formal language, which comprises a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms.
- I kind of dislike this definition.

A Few More

- **Compiler** - a computer program that translates computer code written in one programming language (the source language) into another language (the target language).
- **Computable Function** - A function that can be represented by a *general recursive function*, a **lambda computable function**, or a **Turing Machine**
- **Programming Language** - a formal language, which comprises a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms.
- I kind of dislike this definition.
- My definition - A formal language that has a computable semantics specifying the denotation of a “sentence” written in the language. This denotation can be given by compilation into some other formal language, or by a direct interpretation.

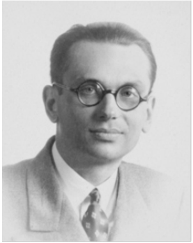
Some Important People


Kurt Gödel

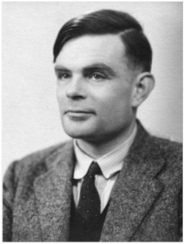

Alan Turing


Alonzo Church

Some Important People



Kurt Gödel

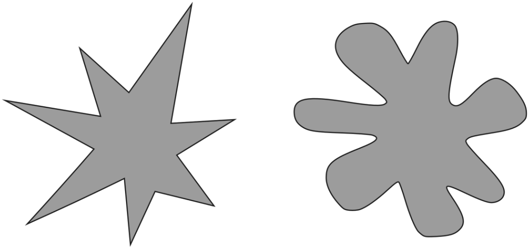


Alan Turing

Alonzo Church

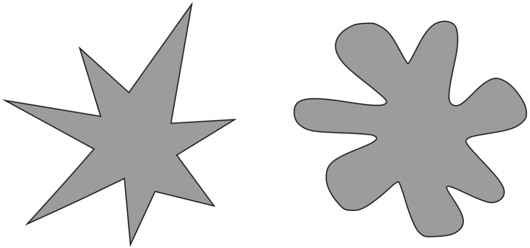
Natural Interpretations?

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



Natural Interpretations?

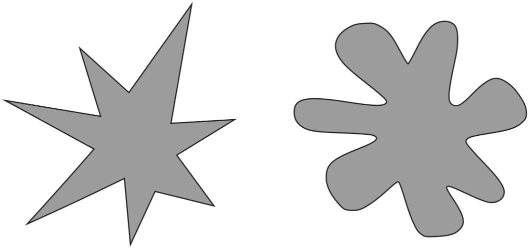
Here's a fun little experiment. Look at these two shapes:



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

Natural Interpretations?

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*.

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.

```
(struct ball [location velocity] \#:transparent)
; 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.

  ```
  (struct ball [location velocity] \#:transparent)
  ; 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

(struct vel [deltax deltay] \#:transparent)
; A Vel is a structure:
;    (vel Number Number)
; interpretation (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 $\to$ Number
- Point Vec $\to$ 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 → Number
- Point Vec → Point
- Point → 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.

Racket Data and the Universe

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

Racket 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.

Racket 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.
```

Racket Data and the Universe

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

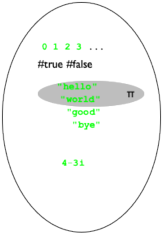
- From here, we talked about enumerations, intervals, and itemizations as a way to restrict the kind of values from the universe that we want our function to consider.
- For example, look at this data definition:

```
; 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.

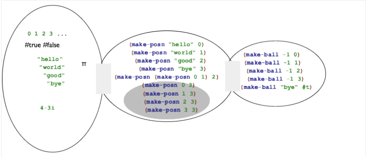
- 

- 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 (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.
- 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.

Back to the Drawing Board

As usual, introducing structs necessitates that we change our design process so that we have clear instructions on how to deal with structs as the input and output to/of functions. We will illustrate the changes with the following problem:

Back to the Drawing Board

As usual, introducing structs necessitates that we change our design process so that we have clear instructions on how to deal with structs as the input and output to/of functions. We will illustrate the changes with the following problem:

- **Sample Problem** Design a function that computes the distance of objects in a 3-dimensional space to the origin.

Back to the Drawing Board

As usual, introducing structs necessitates that we change our design process so that we have clear instructions on how to deal with structs as the input and output to/of functions. We will illustrate the changes with the following problem:

- **Sample Problem** Design a function that computes the distance of objects in a 3-dimensional space to the origin.
- The first step to our design process has a bigger change than was typical for previous revisions to our design process.

Step 1 and Structs

Step 1 and Structs

- (1) When a problem calls for the representation of pieces of information that belong together or describe a natural whole, you need a structure type definition. It requires as many fields as there are relevant properties. An instance of this structure type corresponds to the whole, and the values in the fields correspond to its attributes.

Step 1 and Structs

- (1) When a problem calls for the representation of pieces of information that belong together or describe a natural whole, you need a structure type definition. It requires as many fields as there are relevant properties. An instance of this structure type corresponds to the whole, and the values in the fields correspond to its attributes.
- (1) A data definition for a structure type introduces a name for the collection of instances that are legitimate. Furthermore, it must describe which kind of data goes with which field. Use only names of built-in data collections or previously defined data definitions.

Step 1 and Structs

- (1) When a problem calls for the representation of pieces of information that belong together or describe a natural whole, you need a structure type definition. It requires as many fields as there are relevant properties. An instance of this structure type corresponds to the whole, and the values in the fields correspond to its attributes.
- (1) A data definition for a structure type introduces a name for the collection of instances that are legitimate. Furthermore, it must describe which kind of data goes with which field. Use only names of built-in data collections or previously defined data definitions.
- (1) In the end, we (and others) must be able to use the data definition to create sample structure instances. Otherwise, something is wrong with our data definition. To ensure that we can create instances, our data definitions should come with data examples.

Add Steps 2 and 3

Here's how we apply our step 1 changes to our sample problem:

```
(struct r3 [x y z] \#:transparent)
; An R3 is a structure:
;   (r3 Number Number Number)

(define ex1 (r3 1 2 13))
(define ex2 (r3 -1 0 3))
```

Add Steps 2 and 3

Here's how we apply our step 1 changes to our sample problem:

```
(struct r3 [x y z] \#:transparent)
; An R3 is a structure:
;   (r3 Number Number Number)

(define ex1 (r3 1 2 13))
(define ex2 (r3 -1 0 3))
```

- (2) You still need a signature, a purpose statement, and a function header but the process remains the same.

Add Steps 2 and 3

Here's how we apply our step 1 changes to our sample problem:

```
(struct r3 [x y z] \#:transparent)
; An R3 is a structure:
;   (r3 Number Number Number)

(define ex1 (r3 1 2 13))
(define ex2 (r3 -1 0 3))
```

- (2) You still need a signature, a purpose statement, and a function header but the process remains the same.
- (2) Consider:

```
;;R3 -> Number
;;Computes the distance of a R3 triple to the origin
(define (r3-distance triple) 0)
```

Add Steps 2 and 3

Here's how we apply our step 1 changes to our sample problem:

```
(struct r3 [x y z] \#:transparent)
; An R3 is a structure:
;   (r3 Number Number Number)

(define ex1 (r3 1 2 13))
(define ex2 (r3 -1 0 3))
```

- (2) You still need a signature, a purpose statement, and a function header but the process remains the same.
- (2) Consider:

```
;;R3 -> Number
;;Computes the distance of a R3 triple to the origin
(define (r3-distance triple) 0)
```

- (3) Your functional examples should use the examples generated by step (1):

```
(check-expect (r3-distance ex1) (sqrt 174))
(check-expect (r3-distance ex2) (sqrt 10))
```

Finishing The Steps

We've talked about changes to taking inventory previously:

Finishing The Steps

We've talked about changes to taking inventory previously:

- (4) A function that consumes structures usually–though not always–extracts the values from the various fields in the structure. To remind yourself of this possibility, add a selector for each field to the templates for such functions. For nested structures, consider extracting a single field per nested structure. If necessary comment next to a selector what kind of value it extracts.

Finishing The Steps

We've talked about changes to taking inventory previously:

- (4) A function that consumes structures usually–though not always–extracts the values from the various fields in the structure. To remind yourself of this possibility, add a selector for each field to the templates for such functions. For nested structures, consider extracting a single field per nested structure. If necessary comment next to a selector what kind of value it extracts.
- (5) Use the selector expressions from the template when you define the function. Delete unneeded selections.

Finishing The Steps

We've talked about changes to taking inventory previously:

- (4) A function that consumes structures usually–though not always–extracts the values from the various fields in the structure. To remind yourself of this possibility, add a selector for each field to the templates for such functions. For nested structures, consider extracting a single field per nested structure. If necessary comment next to a selector what kind of value it extracts.
- (5) Use the selector expressions from the template when you define the function. Delete unneeded selections.
- (6) As usual, except that your tests are based on examples from step (1) and other examples.

Language and Cognition

Disclaimer: I'm neither a cognitive scientist, a linguist, a psychologist, nor a philosopher.

Language and Cognition

Disclaimer: I'm neither a cognitive scientist, a linguist, a psychologist, nor a philosopher.

- There was a popular theory, the *Sapir-Whorf Hypothesis*, that stated that language affected our thought processes at a fundamental level. This is linguistic determinism.

Language and Cognition

Disclaimer: I'm neither a cognitive scientist, a linguist, a psychologist, nor a philosopher.

- There was a popular theory, the *Sapir-Whorf Hypothesis*, that stated that language affected our thought processes at a fundamental level. This is linguistic determinism.
- The strong version says that language determines thought and that linguistic categories limit and *determine* cognitive categories.

Language and Cognition

Disclaimer: I'm neither a cognitive scientist, a linguist, a psychologist, nor a philosopher.

- There was a popular theory, the *Sapir-Whorf Hypothesis*, that stated that language affected our thought processes at a fundamental level. This is linguistic determinism.
- The strong version says that language determines thought and that linguistic categories limit and *determine* cognitive categories.
- The weak version says that linguistic categories and usage only *influence* thought and decisions.

Language and Cognition

Disclaimer: I'm neither a cognitive scientist, a linguist, a psychologist, nor a philosopher.

- There was a popular theory, the *Sapir-Whorf Hypothesis*, that stated that language affected our thought processes at a fundamental level. This is linguistic determinism.
- The strong version says that language determines thought and that linguistic categories limit and *determine* cognitive categories.
- The weak version says that linguistic categories and usage only *influence* thought and decisions.
- Do you find that plausible? What are the implications?

Language and Cognition

Disclaimer: I'm neither a cognitive scientist, a linguist, a psychologist, nor a philosopher.

- There was a popular theory, the *Sapir-Whorf Hypothesis*, that stated that language affected our thought processes at a fundamental level. This is linguistic determinism.
- The strong version says that language determines thought and that linguistic categories limit and *determine* cognitive categories.
- The weak version says that linguistic categories and usage only *influence* thought and decisions.
- Do you find that plausible? What are the implications?
- The strong version has been rejected, but the weaker version has been supported. Could something similar apply to the first programming language you learn?

Language and Cognition (cont.)

Noam Chomsky was a big opponent of linguistic determinism in the strong sense.

Language and Cognition (cont.)

Noam Chomsky was a big opponent of linguistic determinism in the strong sense.

- There are other similar arguments in philosophy.

Language and Cognition (cont.)

Noam Chomsky was a big opponent of linguistic determinism in the strong sense.

- There are other similar arguments in philosophy.
- One interesting thing to think about is that our mathematical theorems depend on what we accept as axioms.

Language and Cognition (cont.)

Noam Chomsky was a big opponent of linguistic determinism in the strong sense.

- There are other similar arguments in philosophy.
- One interesting thing to think about is that our mathematical theorems depend on what we accept as axioms.
- Back to the main point, if you learned Racket before Java or Python, would you bother to ask questions about inheritance, overloading, ...?

Language and Cognition (cont.)

Noam Chomsky was a big opponent of linguistic determinism in the strong sense.

- There are other similar arguments in philosophy.
- One interesting thing to think about is that our mathematical theorems depend on what we accept as axioms.
- Back to the main point, if you learned Racket before Java or Python, would you bother to ask questions about inheritance, overloading, ...?
- Our first programming language affects the questions about the ones we learn later.

Language and Cognition (cont.)

Noam Chomsky was a big opponent of linguistic determinism in the strong sense.

- There are other similar arguments in philosophy.
- One interesting thing to think about is that our mathematical theorems depend on what we accept as axioms.
- Back to the main point, if you learned Racket before Java or Python, would you bother to ask questions about inheritance, overloading, ...?
- Our first programming language affects the questions about the ones we learn later.
- We tend to back translate new concepts into languages that we are familiar with.

Language and Cognition (cont.)

Noam Chomsky was a big opponent of linguistic determinism in the strong sense.

- There are other similar arguments in philosophy.
- One interesting thing to think about is that our mathematical theorems depend on what we accept as axioms.
- Back to the main point, if you learned Racket before Java or Python, would you bother to ask questions about inheritance, overloading, ...?
- Our first programming language affects the questions about the ones we learn later.
- We tend to back translate new concepts into languages that we are familiar with.
- Now returning to Racket, let's think about how we design big-bang applications around structs.

The World is a Structure

In a previous program, we used the `Point` struct as the state of the world.

The World is a Structure

In a previous program, we used the `Point` struct as the state of the world.

- In general, for each piece of state that we need to track in the world, we create a struct containing that piece of state

The World is a Structure

In a previous program, we used the `Point` struct as the state of the world.

- In general, for each piece of state that we need to track in the world, we create a struct containing that piece of state
- Let's say we have a Space Invaders style game, where a player controls a tank that must shut down a UFO.

The World is a Structure

In a previous program, we used the `Point` struct as the state of the world.

- In general, for each piece of state that we need to track in the world, we create a struct containing that piece of state
- Let's say we have a Space Invaders style game, where a player controls a tank that must shut down a UFO.
- Our state becomes
  ```
  (struct space-game [ufo tank] \#:transparent)
  ```

The World is a Structure

In a previous program, we used the `Point` struct as the state of the world.

- In general, for each piece of state that we need to track in the world, we create a struct containing that piece of state
- Let's say we have a Space Invaders style game, where a player controls a tank that must shut down a UFO.
- Our state becomes
  ```
  (struct space-game [ufo tank] \#:transparent)
  ```
- Let's come up with a data interpretation. Let's assume the ufo descends 2-dimensionally with random jumps to the left or right.

Space Game State

The state for our space game is relatively straightforward:

Space Game State

The state for our space game is relatively straightforward:

```
; A SpaceGame is a structure:
;   (space-game Point Number).
; interpretation (space-game (point ux uy) tx)
; describes a configuration where the UFO is
; at (ux,uy) and the tank's x-coordinate is tx
```

-

Space Game State

The state for our space game is relatively straightforward:

```
; A SpaceGame is a structure:
;    (space-game Point Number).
; interpretation (space-game (point ux uy) tx)
; describes a configuration where the UFO is
; at (ux,uy) and the tank's x-coordinate is tx
```

-
- What would we need if we wanted to store lots of enemies and not just a single UFO?

Space Game State

The state for our space game is relatively straightforward:

```
; A SpaceGame is a structure:
;   (space-game Point Number).
; interpretation (space-game (point ux uy) tx)
; describes a configuration where the UFO is
; at (ux,uy) and the tank's x-coordinate is tx
```

-
- What would we need if we wanted to store lots of enemies and not just a single UFO?
- Also think about how we would randomly add enemies to a scene in a Geometry Wars style game.

Space Game State

The state for our space game is relatively straightforward:

```
; A SpaceGame is a structure:
;   (space-game Point Number).
; interpretation (space-game (point ux uy) tx)
; describes a configuration where the UFO is
; at (ux,uy) and the tank's x-coordinate is tx
```

-
- What would we need if we wanted to store lots of enemies and not just a single UFO?
- Also think about how we would randomly add enemies to a scene in a Geometry Wars style game.
- Continue to think about other kinds of games and what kind of state we need.

Making a Text Editor

We've been thinking about games and positions of game objects too much, so let's not get hung up on these kinds of structures.

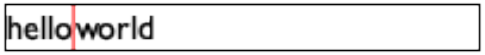
Making a Text Editor

We've been thinking about games and positions of game objects too much, so let's not get hung up on these kinds of structures.

- Let's imagine a struct for the state of a text editor that looks like the following:

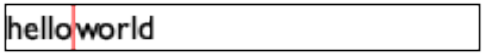
Making a Text Editor

We've been thinking about games and positions of game objects too much, so let's not get hung up on these kinds of structures.

- Let's imagine a struct for the state of a text editor that looks like the following:
- 

Making a Text Editor

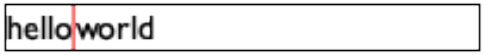
We've been thinking about games and positions of game objects too much, so let's not get hung up on these kinds of structures.

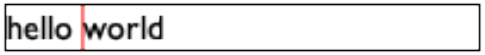
- Let's imagine a struct for the state of a text editor that looks like the following:
- 

- If we press space it updates to the following:

Making a Text Editor

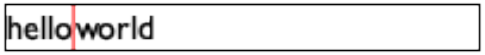
We've been thinking about games and positions of game objects too much, so let's not get hung up on these kinds of structures.

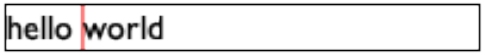
- Let's imagine a struct for the state of a text editor that looks like the following:
- 

- If we press space it updates to the following:
- 

Making a Text Editor

We've been thinking about games and positions of game objects too much, so let's not get hung up on these kinds of structures.

- Let's imagine a struct for the state of a text editor that looks like the following:
- 

- If we press space it updates to the following:
- 

- Here is a start to our data interpretation:

```
(struct editor [pre post] \#:transparent)
; An Editor is a structure:
;    (editor String String)
; interpretation (editor s t) describes an editor
; whose visible text is (string-append s t) with
; the cursor displayed between s and t
```

Making a Text Editor (cont.)

So, we are creating an editor structure that tracks the text before and after the cursor.

Making a Text Editor (cont.)

So, we are creating an editor structure that tracks the text before and after the cursor.

- Normal key input inserts characters in the "pre" string.

Making a Text Editor (cont.)

So, we are creating an editor structure that tracks the text before and after the cursor.

- Normal key input inserts characters in the "pre" string.
- Backspace removes a character from the "pre" string.

Making a Text Editor (cont.)

So, we are creating an editor structure that tracks the text before and after the cursor.

- Normal key input inserts characters in the “pre” string.
- Backspace removes a character from the “pre” string.
- How do we start writing code that manipulates the “pre” string?

Making a Text Editor (cont.)

So, we are creating an editor structure that tracks the text before and after the cursor.

- Normal key input inserts characters in the “pre” string.
- Backspace removes a character from the “pre” string.
- How do we start writing code that manipulates the “pre” string?
- `(editor-pre ed)`

Making a Text Editor (cont.)

So, we are creating an editor structure that tracks the text before and after the cursor.

- Normal key input inserts characters in the "pre" string.
- Backspace removes a character from the "pre" string.
- How do we start writing code that manipulates the "pre" string?
- `(editor-pre ed)`
- What kind of function have we been using to build new strings from old ones?

Making a Text Editor (cont.)

So, we are creating an editor structure that tracks the text before and after the cursor.

- Normal key input inserts characters in the "pre" string.
- Backspace removes a character from the "pre" string.
- How do we start writing code that manipulates the "pre" string?
- `(editor-pre ed)`
- What kind of function have we been using to build new strings from old ones?
- `string-append`

Making a Text Editor (cont.)

So, we are creating an editor structure that tracks the text before and after the cursor.

- Normal key input inserts characters in the “pre” string.
- Backspace removes a character from the “pre” string.
- How do we start writing code that manipulates the “pre” string?
- `(editor-pre ed)`
- What kind of function have we been using to build new strings from old ones?
- `string-append`
- What kind of function have we been using to take apart strings?

Making a Text Editor (cont.)

So, we are creating an editor structure that tracks the text before and after the cursor.

- Normal key input inserts characters in the “pre” string.
- Backspace removes a character from the “pre” string.
- How do we start writing code that manipulates the “pre” string?
- `(editor-pre ed)`
- What kind of function have we been using to build new strings from old ones?
- `string-append`
- What kind of function have we been using to take apart strings?
- `substring`

String Commandments

Here are the first three commandments of functions that take in strings and produce strings:

String Commandments

Here are the first three commandments of functions that take in strings and produce strings:

1. Building a new string from old ones? Thou shalt use `string-append`.

String Commandments

Here are the first three commandments of functions that take in strings and produce strings:

1. Building a new string from old ones? Thou shalt use `string-append`.
2. Building a new string from certain parts of an old one? Thou shalt isolate these parts with `substring`.

String Commandments

Here are the first three commandments of functions that take in strings and produce strings:

1. Building a new string from old ones? Thou shalt use `string-append`.
2. Building a new string from certain parts of an old one? Thou shalt isolate these parts with `substring`.
3. Need to isolate a specific character form a string? Thou shalt use `string-ref` to extract this character?

Changing Representations

What happens if the state of the world for our editor programs takes on a different shape?

Changing Representations

What happens if the state of the world for our editor programs takes on a different shape?

- Consider representing our Editor like this:

Changing Representations

What happens if the state of the world for our editor programs takes on a different shape?

- Consider representing our Editor like this:

```
(struct editor [text location] \#:transparent)
; An Editor is a structure:
;   (editor String Number)
; interpretation (editor s t) describes an editor
; whose visible text is inside of text
; the cursor displayed between index location and location-1 in
; a zero-indexed string
```

-

Changing Representations

What happens if the state of the world for our editor programs takes on a different shape?

- Consider representing our Editor like this:

```
(struct editor [text location] \#:transparent)
; An Editor is a structure:
;   (editor String Number)
; interpretation (editor s t) describes an editor
; whose visible text is inside of text
; the cursor displayed between index location and location-1 in
; a zero-indexed string
```

-
- What changes does this give to how we reason about the program?

Changing Representations

What happens if the state of the world for our editor programs takes on a different shape?

- Consider representing our Editor like this:

```
(struct editor [text location] \#:transparent)
; An Editor is a structure:
;   (editor String Number)
; interpretation (editor s t) describes an editor
; whose visible text is inside of text
; the cursor displayed between index location and location-1 in
; a zero-indexed string
```

-
- What changes does this give to how we reason about the program?
- **Advantage**: the arrow key events become much simpler!

Changing Representations

What happens if the state of the world for our editor programs takes on a different shape?

- Consider representing our Editor like this:

```
(struct editor [text location] \#:transparent)
; An Editor is a structure:
;   (editor String Number)
; interpretation (editor s t) describes an editor
; whose visible text is inside of text
; the cursor displayed between index location and location-1 in
; a zero-indexed string
```

-
- What changes does this give to how we reason about the program?
- **Advantage**: the arrow key events become much simpler!
- **Disadvantage**: For regular key presses, updating the string and the cursor index becomes slightly more involved.

Changing Representations

What happens if the state of the world for our editor programs takes on a different shape?

- Consider representing our Editor like this:

```
(struct editor [text location] \#:transparent)
; An Editor is a structure:
;   (editor String Number)
; interpretation (editor s t) describes an editor
; whose visible text is inside of text
; the cursor displayed between index location and location-1 in
; a zero-indexed string
```

-
- What changes does this give to how we reason about the program?
- **Advantage**: the arrow key events become much simpler!
- **Disadvantage**: For regular key presses, updating the string and the cursor index becomes slightly more involved.
- How does this relate to Sapir-Whorf?

World States and Temporary Objects

What happens to programs where certain objects have a lifespan?

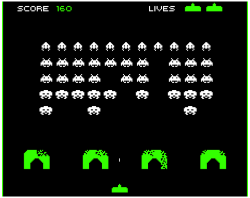
World States and Temporary Objects

What happens to programs where certain objects have a lifespan?

- Consider a space invaders like game where we can shoot a *single* bullet at a time at a *single enemy*.

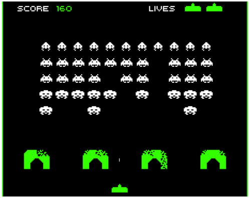
World States and Temporary Objects

What happens to programs where certain objects have a lifespan?

- Consider a space invaders like game where we can shoot a *single* bullet at a time at a *single enemy*.
- 

World States and Temporary Objects

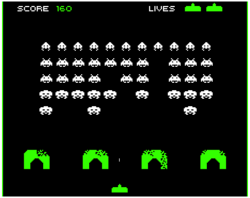
What happens to programs where certain objects have a lifespan?

- Consider a space invaders like game where we can shoot a *single* bullet at a time at a *single enemy*.
- 

- At its most complicated our program only has three objects.

World States and Temporary Objects

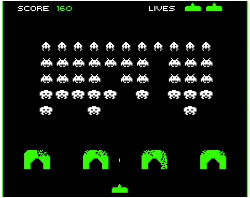
What happens to programs where certain objects have a lifespan?

- Consider a space invaders like game where we can shoot a *single* bullet at a time at a *single enemy*.
- 

- At its most complicated our program only has three objects.
- It has the player vehicle, an enemy space ship, and a bullet.

World States and Temporary Objects

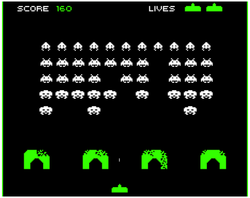
What happens to programs where certain objects have a lifespan?

- Consider a space invaders like game where we can shoot a *single* bullet at a time at a *single enemy*.
- 

- At its most complicated our program only has three objects.
- It has the player vehicle, an enemy space ship, and a bullet.
- When the player doesn't shoot there are only two things, the enemy ship and the player ship.

World States and Temporary Objects

What happens to programs where certain objects have a lifespan?

- Consider a space invaders like game where we can shoot a *single* bullet at a time at a *single enemy*.
- 

- At its most complicated our program only has three objects.
- It has the player vehicle, an enemy space ship, and a bullet.
- When the player doesn't shoot there are only two things, the enemy ship and the player ship.
- We can consider the state where a bullet destroys the enemy ship as the exit condition and so our world state doesn't have to worry about a state where the enemy ship isn't present.

Scalability?

Our state of the world became an itemization of different structs. As we consider adding a score, levels, etc. how must our data definition for the state of the world change?

Scalability?

Our state of the world became an itemization of different structs. As we consider adding a score, levels, etc. how must our data definition for the state of the world change?

- What happens to our code?

Scalability?

Our state of the world became an itemization of different structs. As we consider adding a score, levels, etc. how must our data definition for the state of the world change?

- What happens to our code?
- We have to cond on different massive structs.

Scalability?

Our state of the world became an itemization of different structs. As we consider adding a score, levels, etc. how must our data definition for the state of the world change?

- What happens to our code?
- We have to cond on different massive structs.
- Is having multiple god-structs really a nice way to structure our code?

Scalability?

Our state of the world became an itemization of different structs. As we consider adding a score, levels, etc. how must our data definition for the state of the world change?

- What happens to our code?
- We have to cond on different massive structs.
- Is having multiple god-structs really a nice way to structure our code?
- Even in object oriented programs, it's typical to have god objects and it's better than having hidden state across many different modules.

Scalability?

Our state of the world became an itemization of different structs. As we consider adding a score, levels, etc. how must our data definition for the state of the world change?

- What happens to our code?
- We have to cond on different massive structs.
- Is having multiple god-structs really a nice way to structure our code?
- Even in object oriented programs, it's typical to have god objects and it's better than having hidden state across many different modules.
- Just ensure that the god object defers functionality to other modules. And similarly, our god struct is separated from event handling code

Back To Our Space Invaders Game

Let's return to the start of designing our Space Invaders style game.

Back To Our Space Invaders Game

Let's return to the start of designing our Space Invaders style game.

- We mentioned that we needed an Itemization for whether or not a bullet is fired, so let's provide a data interpretation.

Back To Our Space Invaders Game

Let's return to the start of designing our Space Invaders style game.

- We mentioned that we needed an Itemization for whether or not a bullet is fired, so let's provide a data interpretation.

```
; A SIGS is one of:
;   (aim UFO Tank)
;   (fired UFO Tank Missile)
; interpretation represents the complete state of a
; space invader game. Either aiming or fired a shot.
```

-

Back To Our Space Invaders Game

Let's return to the start of designing our Space Invaders style game.

- We mentioned that we needed an Itemization for whether or not a bullet is fired, so let's provide a data interpretation.

```
; A SIGS is one of:
;   (aim UFO Tank)
;   (fired UFO Tank Missile)
; interpretation represents the complete state of a
; space invader game. Either aiming or fired a shot.
```

-
- Now we need to design structs representing our individual objects and also the alternatives of our itemization. Let's start with `aim` and `fired`

Game Structs

The data interpretation for aim is a bit simple, after looking at our itemization.

Game Structs

The data interpretation for aim is a bit simple, after looking at our itemization.

```
(struct aim [ufo tank] #:transparent)
; aim is a structure
; (aim UFO Tank)
; Interpretation, UFO represents the UFO position
; Tank represents the Tank position.
(define ex1 (aim ...))
```

-

Game Structs

The data interpretation for aim is a bit simple, after looking at our itemization.

```
(struct aim [ufo tank] #:transparent)
; aim is a structure
; (aim UFO Tank)
; Interpretation, UFO represents the UFO position
; Tank represents the Tank position.
(define ex1 (aim ...))
```

-
- Similarly, for our design for our fired state

Game Structs

The data interpretation for aim is a bit simple, after looking at our itemization.

```
(struct aim [ufo tank] #:transparent)
; aim is a structure
; (aim UFO Tank)
; Interpretation, UFO represents the UFO position
; Tank represents the Tank position.
(define ex1 (aim ...))
```

-
- Similarly, for our design for our fired state

```
(struct fired [ufo tank missile] #:transparent)
; fired is a structure
; (aim UFO Tank Missile)
; Interpretation, UFO represents the UFO position
; Tank represents the Tank position.
; Missile represents the missile's position
(define ex2 (fired ...))
```

Game Structs (cont.)

Is there something a bit strange about our fired state?

Game Structs (cont.)

Is there something a bit strange about our fired state?

- There is a single Missile struct, what does that imply?

Game Structs (cont.)

Is there something a bit strange about our fired state?

- There is a single Missile struct, what does that imply?
- In other games like Galaga, we can fire more than one bullet at a time. But this is actually true to how Space Invaders plays.

Game Structs (cont.)

Is there something a bit strange about our fired state?

- There is a single Missile struct, what does that imply?
- In other games like Galaga, we can fire more than one bullet at a time. But this is actually true to how Space Invaders plays.
- So, this means that in the fired state, we can't initiate a new fired state, and so there's a *cooldown* period.

Game Structs (cont.)

Is there something a bit strange about our fired state?

- There is a single Missile struct, what does that imply?
- In other games like Galaga, we can fire more than one bullet at a time. But this is actually true to how Space Invaders plays.
- So, this means that in the fired state, we can't initiate a new fired state, and so there's a *cooldown* period.
- Think about how this affects our on-tick event.

Game Structs (cont.)

Is there something a bit strange about our fired state?

- There is a single Missile struct, what does that imply?
- In other games like Galaga, we can fire more than one bullet at a time. But this is actually true to how Space Invaders plays.
- So, this means that in the fired state, we can't initiate a new fired state, and so there's a *cooldown* period.
- Think about how this affects our on-tick event.
- Let's return to designing the rest of our structs for data representation.

Game Structs (cont.)

Does anyone have a natural description for how our UFO data should look?

Game Structs (cont.)

Does anyone have a natural description for how our UFO data should look?

```
; A UFO is a Point
; interpretation (point x y) is the UFO's location
; (using the top-down, left-to-right convention)
```

-

Game Structs (cont.)

Does anyone have a natural description for how our UFO data should look?

```
; A UFO is a Point
; interpretation (point x y) is the UFO's location
; (using the top-down, left-to-right convention)
```

-
- How does a missile struct look?

Game Structs (cont.)

Does anyone have a natural description for how our UFO data should look?

```
; A UFO is a Point
; interpretation (point x y) is the UFO's location
; (using the top-down, left-to-right convention)
```

-
- How does a missile struct look?

```
; A Missile is a Point.
; interpretation (point x y) is the missile's place
```

-

Game Structs (cont.)

Does anyone have a natural description for how our UFO data should look?

```
; A UFO is a Point
; interpretation (point x y) is the UFO's location
; (using the top-down, left-to-right convention)
```

-
- How does a missile struct look?

```
; A Missile is a Point.
; interpretation (point x y) is the missile's place
```

-
- Is our player vehicle (tank) more complicated than the UFO, what does its data description look like?

Game Structs (cont.)

Does anyone have a natural description for how our UFO data should look?

```
; A UFO is a Point
; interpretation (point x y) is the UFO's location
; (using the top-down, left-to-right convention)
```

-
- How does a missile struct look?

```
; A Missile is a Point.
; interpretation (point x y) is the missile's place
```

-
- Is our player vehicle (tank) more complicated than the UFO, what does its data description look like?

```
(struct tank [loc vel] #:transparent)
; A Tank is a structure:
;   (tank Number Number).
; interpretation (tank x dx) specifies the position:
; (x, HEIGHT) and the tank's speed: dx pixels/tick
```

Game Structs (cont.)

Can anyone explain why we defined a new Tank struct, while we left the UFO and Missile data interpretations as being aliases for points?

Game Structs (cont.)

Can anyone explain why we defined a new Tank struct, while we left the UFO and Missile data interpretations as being aliases for points?

- Our tank had a movement vector (which can contain negative values) associated with it, while the other two just have a position that is updated.

Game Structs (cont.)

Can anyone explain why we defined a new Tank struct, while we left the UFO and Missile data interpretations as being aliases for points?

- Our tank had a movement vector (which can contain negative values) associated with it, while the other two just have a position that is updated.
- This returns to our discussion of making sure our data lives in the correct domain with a clear interpretation.

Game Structs (cont.)

Can anyone explain why we defined a new Tank struct, while we left the UFO and Missile data interpretations as being aliases for points?

- Our tank had a movement vector (which can contain negative values) associated with it, while the other two just have a position that is updated.
- This returns to our discussion of making sure our data lives in the correct domain with a clear interpretation.
- Now that we are done with specifying data interpretations, what kind data examples can we give for our SIGS itemization?

Game Struct (cont.)

```
;;Aiming and moving left
(define ex1 (aim (point 20 10) (tank 28 -3)))

;;Just fired
(define ex2 (fired (point 20 10)
                   (tank 28 -3)
                   (point 28 (- HEIGHT TANK-HEIGHT))))

;;Missile is about to collide with UFO
(define ex3 (fired (point 20 100)
                   (tank 100 3)
                   (point 22 103)))
```

Game Struct (cont.)

```
;;Aiming and moving left
(define ex1 (aim (point 20 10) (tank 28 -3)))

;;Just fired
(define ex2 (fired (point 20 10)
            (tank 28 -3)
            (point 28 (- HEIGHT TANK-HEIGHT))))

;;Missile is about to collide with UFO
(define ex3 (fired (point 20 100)
            (tank 100 3)
            (point 22 103)))
```

- We only had two states for our itemization, why did we need three examples?

Game Struct (cont.)

```
;;Aiming and moving left
(define ex1 (aim (point 20 10) (tank 28 -3)))

;;Just fired
(define ex2 (fired (point 20 10)
            (tank 28 -3)
            (point 28 (- HEIGHT TANK-HEIGHT))))

;;Missile is about to collide with UFO
(define ex3 (fired (point 20 100)
            (tank 100 3)
            (point 22 103)))
```

- We only had two states for our itemization, why did we need three examples?
- We had an *implicit* state arising from the fired state.

Game Struct (cont.)

```
;;Aiming and moving left
(define ex1 (aim (point 20 10) (tank 28 -3)))

;;Just fired
(define ex2 (fired (point 20 10)
             (tank 28 -3)
             (point 28 (- HEIGHT TANK-HEIGHT))))

;;Missile is about to collide with UFO
(define ex3 (fired (point 20 100)
             (tank 100 3)
             (point 22 103)))
```

- We only had two states for our itemization, why did we need three examples?
- We had an *implicit* state arising from the fired state.
- Can we think of another implicit state?

Game Struct (cont.)

```
;;Aiming and moving left
(define ex1 (aim (point 20 10) (tank 28 -3)))

;;Just fired
(define ex2 (fired (point 20 10)
            (tank 28 -3)
            (point 28 (- HEIGHT TANK-HEIGHT))))

;;Missile is about to collide with UFO
(define ex3 (fired (point 20 100)
            (tank 100 3)
            (point 22 103)))
```

- We only had two states for our itemization, why did we need three examples?
- We had an *implicit* state arising from the fired state.
- Can we think of another implicit state?
- Yes, when the bullet goes off screen.

Game Struct (cont.)

```
;;Aiming and moving left
(define ex1 (aim (point 20 10) (tank 28 -3)))

;;Just fired
(define ex2 (fired (point 20 10)
            (tank 28 -3)
            (point 28 (- HEIGHT TANK-HEIGHT))))

;;Missile is about to collide with UFO
(define ex3 (fired (point 20 100)
            (tank 100 3)
            (point 22 103)))
```

- We only had two states for our itemization, why did we need three examples?
- We had an *implicit* state arising from the fired state.
- Can we think of another implicit state?
- Yes, when the bullet goes off screen.
- Certain states shouldn't require new data definitions but must be handled!

Design Process 3.0

Now that we are considering modeling pieces of related state with structs and our overall possible event structures with itemizations, let's discuss the last changes to our design process for this section of the book.

Design Process 3.0

Now that we are considering modeling pieces of related state with structs and our overall possible event structures with itemizations, let's discuss the last changes to our design process for this section of the book.

- (1) An itemization of different forms of dataincluding collections of structuresis required when your problem statement distinguishes different kinds of information and when at least some of these pieces of information consist of several different pieces.

Design Process 3.0

Now that we are considering modeling pieces of related state with structs and our overall possible event structures with itemizations, let's discuss the last changes to our design process for this section of the book.

- (1) An itemization of different forms of dataincluding collections of structuresis required when your problem statement distinguishes different kinds of information and when at least some of these pieces of information consist of several different pieces.
- Steps (2) and (3) remain the same.

Design Process 3.0

Now that we are considering modeling pieces of related state with structs and our overall possible event structures with itemizations, let's discuss the last changes to our design process for this section of the book.

- (1) An itemization of different forms of dataincluding collections of structuresis required when your problem statement distinguishes different kinds of information and when at least some of these pieces of information consist of several different pieces.
- Steps (2) and (3) remain the same.
- (4) Write a cond for dealing with itemization input and have a predicate for each data from in the itemization. If one of the items in the itemization is a struct, add selectors. With a separate data definition, for items, instead of writing selectors, write a separate function.

Design Process 3.0 (cont.)

- (5) Fill out the code, starting with the easy parts. If dealing with an itemization and you're stuck, analyze the functional examples dealing with this item. If your template calls another templated function, assume this function works as expected and continue.

Design Process 3.0 (cont.)

- (5) Fill out the code, starting with the easy parts. If dealing with an itemization and you're stuck, analyze the functional examples dealing with this item. If your template calls another templated function, assume this function works as expected and continue.
- (6) Same as before.

Checking the Input Types

Although we can usually trust that internally used functions receive input values of the proper type, we occasionally have to input values for API style functions.

Checking the Input Types

Although we can usually trust that internally used functions receive input values of the proper type, we occasionally have to input values for API style functions.

- Consider writing a function that computes the area of a disk when given numbers and errors for other input types.

Checking the Input Types

Although we can usually trust that internally used functions receive input values of the proper type, we occasionally have to input values for API style functions.

- Consider writing a function that computes the area of a disk when given numbers and errors for other input types.

  ```
  ; Any -> Number
  ; computes the area of a disk with radius v,
  ; if v is a number
  (define (checked-area-of-disk v)
    (cond
      [(number? v) (area-of-disk v)]
      [else (error "area-of-disk: number expected")]))
  ```

-

Checking the Input Types

Although we can usually trust that internally used functions receive input values of the proper type, we occasionally have to input values for API style functions.

- Consider writing a function that computes the area of a disk when given numbers and errors for other input types.

```
; Any -> Number
; computes the area of a disk with radius v,
; if v is a number
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [else (error "area-of-disk: number expected")]))
```

-
- For simplicity, we won't write functions that explicitly check against invalid inputs often. And moreover, we will eventually consider rewriting programs in Typed Racket so that such checks are unnecessary.

Checking the Input Types

Although we can usually trust that internally used functions receive input values of the proper type, we occasionally have to input values for API style functions.

- Consider writing a function that computes the area of a disk when given numbers and errors for other input types.

```
; Any -> Number
; computes the area of a disk with radius v,
; if v is a number
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [else (error "area-of-disk: number expected")]))
```

-
- For simplicity, we won't write functions that explicitly check against invalid inputs often. And moreover, we will eventually consider rewriting programs in Typed Racket so that such checks are unnecessary.

Checking the World

The world can get wrong too quickly.

Checking the World

The world can get wrong too quickly.

- Although we agreed that internally used functions get data of the right form, we will check world states since the interplay of handling different events can easily lead to errors.

Checking the World

The world can get wrong too quickly.

- Although we agreed that internally used functions get data of the right form, we will check world states since the interplay of handling different events can easily lead to errors.
- And sometimes an event can produce invalid data that isn't caught until several events later, so it can be confusing to debug.

Checking the World

The world can get wrong too quickly.

- Although we agreed that internally used functions get data of the right form, we will check world states since the interplay of handling different events can easily lead to errors.
- And sometimes an event can produce invalid data that isn't caught until several events later, so it can be confusing to debug.
- To help with this kind of problem, big-bang comes with an optional check-with clause that accepts a predicate for world states.

Checking the World

The world can get wrong too quickly.

- Although we agreed that internally used functions get data of the right form, we will check world states since the interplay of handling different events can easily lead to errors.
- And sometimes an event can produce invalid data that isn't caught until several events later, so it can be confusing to debug.
- To help with this kind of problem, big-bang comes with an optional check-with clause that accepts a predicate for world states.
- If, for example, we chose to represent all world states with Number, we could express this fact easily like this:

```
(define (main s0)
  (big-bang s0 ... [check-with number?] ...))
```

More Complex Checks

This comes in handy when we have more rich restrictions on data, like that our world state should be in some interval.

More Complex Checks

This comes in handy when we have more rich restrictions on data, like that our world state should be in some interval.

- Consider if the world state should be a real number between zero and 1.

More Complex Checks

This comes in handy when we have more rich restrictions on data, like that our world state should be in some interval.

- Consider if the world state should be a real number between zero and 1.
- We can write a predicate that checks this:

```
(define (zero-to-one? num)
  (and (number? num) (<= 0 num 1)))
```

More Complex Checks

This comes in handy when we have more rich restrictions on data, like that our world state should be in some interval.

- Consider if the world state should be a real number between zero and 1.
- We can write a predicate that checks this:

```
(define (zero-to-one? num)
  (and (number? num) (<= 0 num 1)))
```

- Now we can add a check-with clause that uses a predicate to check the output of all world producing events:

```
(define (main s0)
  (big-bang s0
            ...
            [check-with between-0-and-1?]
            ...))
```

Equality Predicates

Let's talk about equality again...at least in programming languages.

Equality Predicates

Let's talk about equality again...at least in programming languages.

- There are certain equalities that a programmer might want to write that errors when applied to values of the correct type that are not in some subset of these values.

Equality Predicates

Let's talk about equality again...at least in programming languages.

- There are certain equalities that a programmer might want to write that errors when applied to values of the correct type that are not in some subset of these values.
- For example, key events and mouse events are itemizations over strings, but `string=?` does not error on strings that are not key events or mouse events

Equality Predicates

Let's talk about equality again...at least in programming languages.

- There are certain equalities that a programmer might want to write that errors when applied to values of the correct type that are not in some subset of these values.
- For example, key events and mouse events are itemizations over strings, but `string=?` does not error on strings that are not key events or mouse events
- Instead, in the future we can use `key=?` and `mouse=?`

Equality Predicates

Let's talk about equality again...at least in programming languages.

- There are certain equalities that a programmer might want to write that errors when applied to values of the correct type that are not in some subset of these values.
- For example, key events and mouse events are itemizations over strings, but `string=?` does not error on strings that are not key events or mouse events
- Instead, in the future we can use `key=?` and `mouse=?`
- We can also define our own equality predicates like `light=?` so that an error is thrown if a string is given that isn't a traffic light.

Equality Predicates

Let's talk about equality again...at least in programming languages.

- There are certain equalities that a programmer might want to write that errors when applied to values of the correct type that are not in some subset of these values.
- For example, key events and mouse events are itemizations over strings, but `string=?` does not error on strings that are not key events or mouse events
- Instead, in the future we can use `key=?` and `mouse=?`
- We can also define our own equality predicates like `light=?` so that an error is thrown if a string is given that isn't a traffic light.
- Let me design something similar

End of Part 1

We have now finished the first part of this book. To summarize what was covered:

End of Part 1

We have now finished the first part of this book. To summarize what was covered:

- Basic Racket syntax and constructs. Defining functions and using conditional expressions.

End of Part 1

We have now finished the first part of this book. To summarize what was covered:

- Basic Racket syntax and constructs. Defining functions and using conditional expressions.
- Designing basic functions in Racket with a design process.

End of Part 1

We have now finished the first part of this book. To summarize what was covered:

- Basic Racket syntax and constructs. Defining functions and using conditional expressions.
- Designing basic functions in Racket with a design process.
- Using different design processes for scripts and event driven programs.

End of Part 1

We have now finished the first part of this book. To summarize what was covered:

- Basic Racket syntax and constructs. Defining functions and using conditional expressions.
- Designing basic functions in Racket with a design process.
- Using different design processes for scripts and event driven programs.
- Using itemizations to specify different kind of values to handle from other existing data types

End of Part 1

We have now finished the first part of this book. To summarize what was covered:

- Basic Racket syntax and constructs. Defining functions and using conditional expressions.
- Designing basic functions in Racket with a design process.
- Using different design processes for scripts and event driven programs.
- Using itemizations to specify different kind of values to handle from other existing data types
- Using structs to group related data and altering our design process for world programs with itemizations and structs