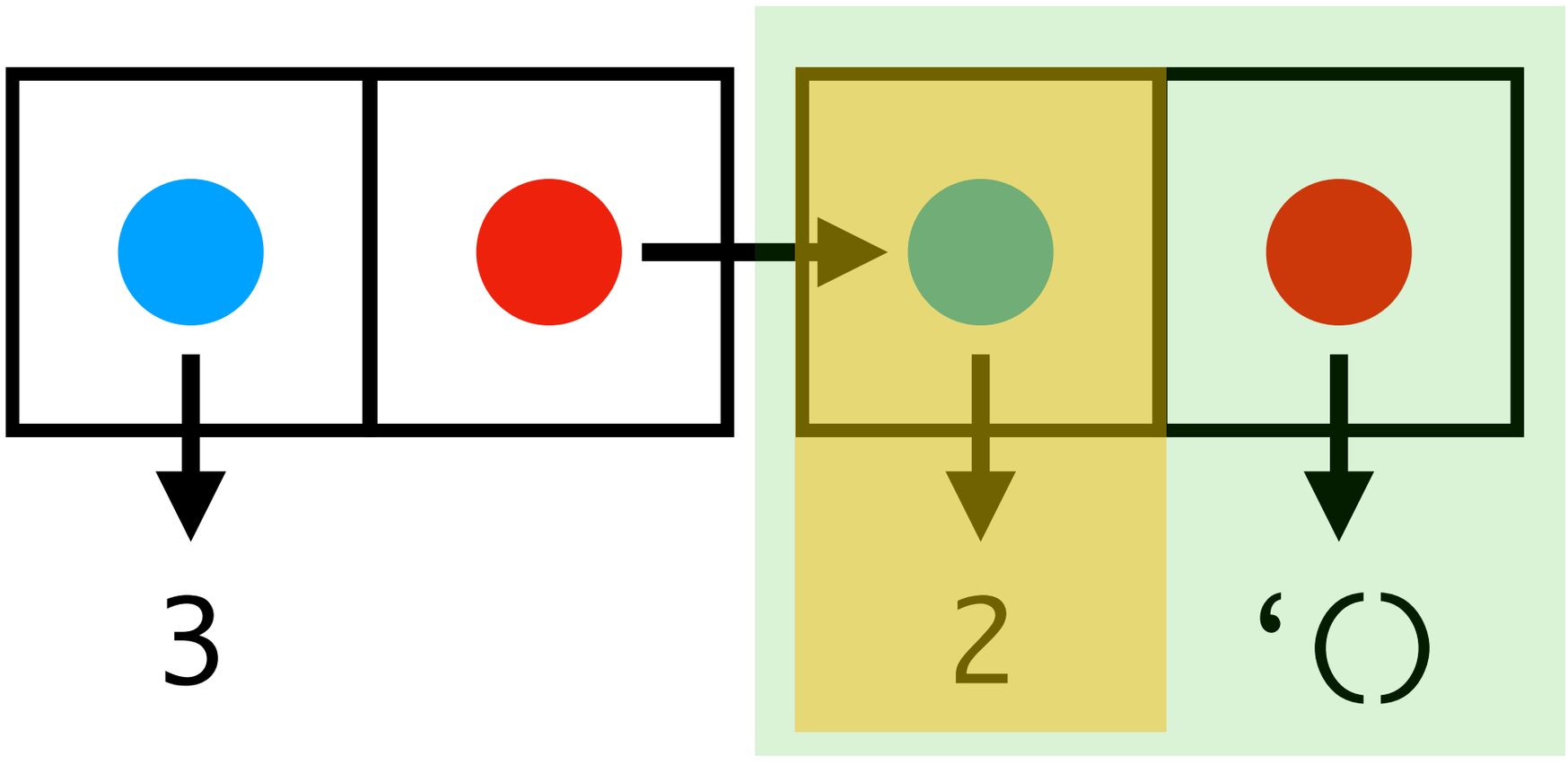
Data Structures in Racket

Part 2

Last time

```
(car
 (cdr
    (cons 3 (cons 2 '()))))
```



This time

5 Programmer-Defined Datatypes

New datatypes are normally created with the `struct` form, which is the topic of this chapter. The class-based object system, which we defer to Classes and Objects, offers an alternate mechanism for creating new datatypes, but even classes and objects are implemented in terms of structure types.

5.1 Simple Structure Types: `struct`

To a first approximation, the syntax of `struct` is

```
(struct struct-id (field-id ...))
```

Examples:

```
(struct posn (x y))
```

The `struct` form binds *struct-id* and a number of identifiers that are built from *struct-id* and the *field-id*s:

- *struct-id* : a *constructor* function that takes as many arguments as the number of *field-id*s, and returns an instance of the structure type.

  Example:

  ```
  > (posn 1 2)
  #<posn>
  ```

- *struct-id*? : a *predicate* function that takes a single argument and returns `#t` if it is an instance of the structure type, `#f` otherwise.

  Examples:

Use **struct** to define a new datatype

```
(struct empty-tree ())
```

```
(struct leaf (elem))
```

```
(struct tree (left right))
```

Copy these

```
(struct empty-tree ())
```

```
(struct leaf (elem))
```

```
(struct tree (value left right))
```

```
(empty-tree)
```

```
(leaf 23)
```

```
(tree 12 (empty-tree) (leaf 23))
```

Racket automatically generates helpers…

`tree?`

`tree-left`

`tree-right`

Write `max-of-tree`

Use the helpers

Pattern matching

Pattern matching allows me to tell Racket the “shape” of what I’m looking for

Manually pulling apart data structures is laborious

```
(define (max-of-tree t)
  (match t
    [(leaf e) e]
    [(tree v _ (empty-tree)) v]
    [(tree _ _ r) (max-of-tree r)]))
```

Variables are bound in the match, refer to in body

```
(define (max-of-tree t)
  (match t
    [(leaf e) e]
    [(tree v _ (empty-tree)) v]
    [(tree _ _ r) (max-of-tree r)]))
```

Note: match struct w/ (name params…)

```
(define (max-of-tree t)
  (match t
    [(leaf e) e]
    [(tree v _ (empty-tree)) v]
    [(tree _ _ r) (max-of-tree r)]))
```

Define `is-sorted`

Can match a list of x's

```
(list x y z …)
```

```
(1 2 3 4)
```

```
x = 1 y = 2 z = ‘(3 4)
```

Can match cons cells too…

```
(cons x y)
```

Variants include things like match-let

10

Racket has a “reader”

(read)

Racket “reads” the input one *datum* at a time

```
> (read)
(1 2 3)
'(1 2 3)
> (read)
1 2 3
1
> (read)
2
> (read)
3
>
```

Read will “buffer” its input

NETFLIX

7%

Loading

```
(read-line)
```

```
(open-input-file)
```

Contracts

```
(define (reverse-string s)
  (list->string (reverse (string->list s))))
```

Write out the call and return type of this for yourself

```
(define (factorial i)
  (cond
    [(= i 1) 1]
    [else (* (factorial (- i 1)) i)]))
```

What are the call / return types?

What is the pre / post condition?

```
(define (gt0? x) (> x 0))
```

```
(define/contract (factorial i)
  (-> gt0? gt0?)
  (cond
    [(= i 1) 1]
    [else (* (factorial (- i 1)) i)]))
```

Now in tail form…

```
(define (fac-tail i)
  (letrec ([h (lambda (i acc)
                (cond
                [(= i 0) acc]
                [else (h (- i 1) (* acc i))]))])
    (h i 1)))
```

Now, let's say I want to say it's equal to factorial…

```
(define/contract (fac-tail i)
  (->i ([x (>=/c 0)])
       [result (x) (lambda (result) (= (factorial x) result))])
  (letrec ([h (lambda (i acc)
                (cond
                  [(= i 0) acc]
                  [else (h (- i 1) (* acc i))]))])
    (h i 1)))
```

```
(->i ([x (>=/c 0)])
     [result (x) (lambda (result) (= (factorial x) result))])
```

```
(define/contract (reverse-string s)
  (-> string? string?)
  (list->string (reverse (string->list s))))
```

```
(define/contract (reverse-string s)
  (-> string? string?)
  (list->string (reverse (string->list s))))
```

(<=/c 2)

<=/c takes an argument x, returns a function f that takes an argument y, and `f(y) = #t` if `x <= y`

<=/c takes an argument x, returns a function f that takes an argument y, and `f(y) = #t` if `x <= y`

(Note: <=/c is also doing some bookeeping, but we won't worry about that now.)

Challenge: write <=/c

Three stories

```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (string-append (f s1) (f s2)))
```

```
(define (reverse-string s)
  (list->string (reverse (string->list s))))
```

Scenario: you call call-and-concat with reverse

Scenario: you call call-and-concat with reverse, 12, and “12"

Now define

```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (length (string-append (f s1) (f s2))))
```

Now define

```
(define/contract (call-and-concat f s1 s2)
(-> (-> string? string?) string? string? string?)
(length (string-append (f s1) (f s2))))
```

What went wrong?

Now define

```
(define/contract (call-and-concat f s1 s2)
(-> (-> string? string?) string? string? string?)
(length (string-append (f s1) (f s2))))
```

What went wrong?

Who is to blame?