Designing Data,
Designing Worlds
17 September 2020
Lab 3, tomorrow

There will be some changes to the lab format

Follow the instructions on the lab webpage
Assignment 1, later today
Academic integrity in Computer Science
What’s appropriate and inappropriate when you’re programming may not always feel clear to you, so I want to go over it.
If you’re told you can work in a pair, e.g., on a lab, it’s fine to collaborate.

But it needs to be a real collaboration; you both need to contribute! Acknowledge your collaborators, e.g., by putting both of your names in a comment.

Exams need to be completed individually.

Unless otherwise instructed, homework assignments need to be completed individually.
Get help, but from the right places.

**People** you should consult:

Me (Campuswire, office hours Zoom)

The coaches for this section of 101 – Alex, Rose, and Juliet (Campuswire, coaching hours Zoom)

**Written resources** you’re always free to consult:

The textbook ([htdp.org](http://htdp.org))

The Racket documentation (from DrRacket’s Help menu or by going to [racket-lang.org](http://racket-lang.org))

Slides, handouts, previous solutions, etc. on our course website ([cs.vassar.edu/~cs101/3](http://cs.vassar.edu/~cs101/3))
Consulting other sources puts you in danger of academic dishonesty (cheating):

On the Internet you might find solutions from other sections of 101, either this semester or in the past courses at other schools random people

Using code and ideas from these sources in your assignments or on an exam is equivalent to copying sentences from someone else’s book, essay, etc. and claiming them as your own.
Our homework assignments are intended for you to think through the problems and, if you get stuck, to talk them through with the coaches or me.

The coaches and I will do our best to help you in a way that leads you to understand \textit{what} to do, rather than \textit{giving you the answers}. Other people and other resources won’t be careful about that, and using someone else’s code or ideas undermines this learning process.
A heuristic for cheating:

Anything that helps you learn is good.
Anything that helps you avoid having to learn is bad.
If you have any questions,

There’s a department guide at cs.vassar.edu/integrity

You can ask on Campuswire (anonymously if you prefer)
It’s important to be ethical when we’re designing computer programs, both in college and beyond.
1.3 Be honest and trustworthy.

Honesty is an essential component of trustworthiness. A computing professional should be transparent and provide full disclosure of all pertinent system capabilities, limitations, and potential problems to the appropriate parties. Making deliberately false or misleading claims, fabricating or falsifying data, offering or accepting bribes, and other dishonest conduct are violations of the Code.

Computing professionals should be honest about their qualifications, and about any limitations in their competence to complete a task. Computing professionals should be forthright about any circumstances that might lead to either real or perceived conflicts of interest or otherwise tend to undermine the independence of their judgment. Furthermore, commitments should be honored.

Computing professionals should not misrepresent an organization's policies or procedures, and should not speak on behalf of an organization unless authorized to do so.
1.5 Respect the work required to produce new ideas, inventions, creative works, and computing artifacts.

Developing new ideas, inventions, creative works, and computing artifacts creates value for society, and those who expend this effort should expect to gain value from their work. Computing professionals should therefore credit the creators of ideas, inventions, work, and artifacts, and respect copyrights, patents, trade secrets, license agreements, and other methods of protecting authors' works.

Both custom and the law recognize that some exceptions to a creator's control of a work are necessary for the public good. Computing professionals should not unduly oppose reasonable uses of their intellectual works. Efforts to help others by contributing time and energy to projects that help society illustrate a positive aspect of this principle. Such efforts include free and open source software and work put into the public domain. Computing professionals should not claim private ownership of work that they or others have shared as public resources.
Recall: Data definitions
Suppose you’re working on a program that someone else wrote for simulating traffic.

You see this function definition:

```
(define (next-color c)
  (cond [(= c 0) 2]
        [(= c 1) 0]
        [(= c 2) 1]))
```

It’s short – but not very clear.
If they’d followed the steps of the design recipe for functions, it would be a little clearer:

;;; next-color : Natural -> Natural
;;; Produce next color of traffic light
(check-expect (next-color 0) 2)
(check-expect (next-color 1) 0)
(check-expect (next-color 2) 1)
(define (next-color c)
  (cond [(= c 0) 2]
        [(= c 1) 0]
        [(= c 2) 1]))
If they’d followed the steps of the design recipe for functions, it would be a little clearer:

```scheme
;; next-color : Natural -> Natural
;; Produce next color of traffic light
(check-expect (next-color 0) 2)
(check-expect (next-color 1) 0)
(check-expect (next-color 2) 1)
(define (next-color c)
  (cond [(= c 0) 2]
        [(= c 1) 0]
        [(= c 2) 1]))
```

What about 3?
If they’d followed the steps of the design recipe for functions, it would be a little clearer:

```scheme
;; next-color : Natural -> Natural
;; Produce next color of traffic light
(check-expect (next-color 0) 2)
(check-expect (next-color 1) 0)
(check-expect (next-color 2) 1)
(define (next-color c)
  (cond
   [(= c 0) 2]
   [(= c 1) 0]
   [(= c 2) 1]))
```

What about 3?

What does it mean for the function to produce 2?
A light is red

...
A *data definition* describes how we are representing information as data.
;; TLColor is one of:
;; - 0
;; - 1
;; - 2
;; Interpretation: color of a traffic light -- 0 is red, 
;; 1 yellow, and is 2 green

Given this data definition, we can write a template for functions that consume a TLColor:

```
(define (tlcolor-template c)
  (cond [(= c 0) ...]
        [(= c 1) ...]
        [(= c 2) ...]))
```
Types of data definitions
Intervals
Write an interval when the information to be represented consists of numbers within a certain range.

*Integer*[0, 10] is all the integers from 0 to 10, inclusive.

*Number*[0, 10) is all the numbers from 0 inclusive to 10 exclusive.

[ and ] include the end point

( and ) do not include the end point
Example

Imagine that you’re designing a program to manage ticket sales for a theater, which happens to be perfectly rectangular.
Example

Imagine that you’re designing a program to manage ticket sales for a theater, which happens to be perfectly rectangular.

*Design a data definition* to represent a seat number in a row, where each row has 32 seats. (Just represent the seat number, not the row number.)
SeatNum is Natural[1, 32]
Interpretation: seat numbers in a row;
1 and 32 are aisle seats
(define SN1 1)
(define SN2 12)
(define SN3 32)
(define (seatnum-template sn)
  (... sn))
;; SeatNum is Natural[1, 32]
;; Interpretation: seat numbers in a row;
;; 1 and 32 are aisle seats
(define SN1 1)
(define SN2 12)
(define SN3 32)
(define (seatnum-template sn)
  (... sn))

Be careful – not Number, which would include decimals!
Example

*Design a function* that produces true if the given seat number is on the aisle.
;; SeatNum is Natural[1, 32]
;; Interpretation: seat numbers in a row;
;; 1 and 32 are aisle seats
(define SN1 1)
(define SN2 12)
(define SN3 32)
(define (seatnum-template sn)
  (... sn))

;; aisle? : SeatNum -> Boolean
;; Produce #true if the given seat number is on the aisle.
(check-expect (aisle? SN1) #true)
(check-expect (aisle? SN2) #false)
(check-expect (aisle? SN3) #true)
;; SeatNum is Natural[1, 32]
;; Interpretation: seat numbers in a row;
;; 1 and 32 are aisle seats
(define SN1 1)
(define SN2 12)
(define SN3 32)
(define (seatnum-template sn)
  (... sn))

;; aisle? : SeatNum -> Boolean
;; Produce #true if the given seat number is on the aisle.
(check-expect (aisle? SN1) #true)
(check-expect (aisle? SN2) #false)
(check-expect (aisle? SN3) #true)
(define aisle? sn)
  (or (= sn 1)
      (= sn 32)))
Enumerations
Write an *enumeration* when the information to be represented consists of a fixed number of distinct items.
Exercise

As part of designing a system to keep track of student grades, you are asked to *design a data definition* to represent the letter grade in a course, which is one of A, B, or C.
LetterGrade is one of:
- "A"
- "B"
- "C"

Interpretation: the letter grade in a course

(define (lettergrade-template lg)
  (cond [(string=? lg "A") ...]
        [(string=? lg "B") ...]
        [(string=? lg "C") ...])))
;; LetterGrade is one of:
;; – "A"
;; – "B"
;; – "C"
;; Interpretation: the letter grade in a course

(define (lettergrade-template lg)
  (cond [(string=? lg "A") ...]
        [(string=? lg "B") ...]
        [(string=? lg "C") ...])))

You could create constant examples, e.g.,
(define LG-A "A"), but they’d just be the
values we listed, so it’s also ok to omit them.
Exercise

*Design a function* that consumes a letter grade and produces the next highest letter grade.
LetterGrade is one of:
- "A"
- "B"
- "C"

Interpretation: the letter grade in a course

(define (lettergrade-template lg)
  (cond [(string=? lg "A") ...]
       [(string=? lg "B") ...]
       [(string=? lg "C") ...])))

bump-up : LetterGrade -> LetterGrade
Produce next highest letter grade (no change for A)
(check-expect (bump-up "A") "A")
(check-expect (bump-up "B") "A")
(check-expect (bump-up "C") "B")
;; LetterGrade is one of:
;; - "A"
;; - "B"
;; - "C"
;; Interpretation: the letter grade in a course
(define (lettergrade-template lg)
  (cond [(string=? lg "A") ...]
        [(string=? lg "B") ...]
        [(string=? lg "C") ...])))

;; bump-up : LetterGrade -> LetterGrade
;; Produce next highest letter grade (no change for A)
(check-expect (bump-up "A") "A")
(check-expect (bump-up "B") "A")
(check-expect (bump-up "C") "B")
(define (bump-up lg)
  (cond [(string=? lg "A") "A"
          [(string=? lg "B") "A"
           [(string=? lg "C") "B"]])))
Traffic light world
All traffic lights are the same size and position on the screen.
All traffic lights are the same size and position on the screen.

What distinguishes them?
All traffic lights are the same size and position on the screen.

What distinguishes them?

Asking this helps us think about data.
All traffic lights are the same size and position on the screen.
All traffic lights are the same size and position on the screen.

*How do we get from one to the other?*
All traffic lights are the same size and position on the screen.

How do we get from one to the other?

Asking this helps us think about functions
Data

- Data definition
- Interpretation
- Examples
- Template

Functions

- Signature
- Purpose
- Tests
- Code
<table>
<thead>
<tr>
<th><strong>Data</strong></th>
<th><strong>Functions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data definition</strong></td>
<td>Signature</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Purpose</td>
</tr>
<tr>
<td>Examples</td>
<td>Tests</td>
</tr>
<tr>
<td>Template</td>
<td>Code</td>
</tr>
</tbody>
</table>
A TrafficLight is one of
A TrafficLight is one of
- "green"
- "yellow"
- "red"
<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data definition</td>
<td>Signature</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Purpose</td>
</tr>
<tr>
<td>Examples</td>
<td>Tests</td>
</tr>
<tr>
<td>Template</td>
<td>Code</td>
</tr>
</tbody>
</table>
A TrafficLight is one of
- "green"
- "yellow"
- "red"

Interpretation: The current color of the light
<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data definition</td>
<td>Signature</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Purpose</td>
</tr>
<tr>
<td>Examples</td>
<td>Tests</td>
</tr>
<tr>
<td>Template</td>
<td>Code</td>
</tr>
</tbody>
</table>
A TrafficLight is one of
- "green"
- "yellow"
- "red"

Interpretation: The current color of
the light

(define TL-GREEN "green")
(define TL-YELLOW "yellow")
(define TL-RED "red")
### Data
- **Data definition**
- **Interpretation**
- **Examples**
- **Template**

### Functions
- **Signature**
- **Purpose**
- **Tests**
- **Code**
A TrafficLight is one of
- "green"
- "yellow"
- "red"

Interpretation: The current color of the light

(define TL-GREEN "green")
(define TL-YELLOW "yellow")
(define TL-RED "red")
A TrafficLight is one of
- "green"
- "yellow"
- "red"

Interpretation: The current color of the light

(define TL-GREEN "green")
(define TL-YELLOW "yellow")
(define TL-RED "red")

(define (trafficlight-template tl)
A TrafficLight is one of
- "green"
- "yellow"
- "red"

Interpretation: The current color of the light

(define TL-GREEN "green")
(define TL-YELLOW "yellow")
(define TL-RED "red")

(define (trafficlight-template tl)
  (cond
   [(string=? tl TL-GREEN) ...]
   [(string=? tl TL-YELLOW) ...]
   [(string=? tl TL-RED) ...])))
<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data definition</td>
<td>Signature</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Purpose</td>
</tr>
<tr>
<td>Examples</td>
<td>Tests</td>
</tr>
<tr>
<td>Template</td>
<td>Code</td>
</tr>
</tbody>
</table>
We have a mechanism for supporting interactive programs, the **big-bang** function.
(big-bang initial-state
   [to-draw draw-function]
   [event-type event-function])
(big-bang initial-state
  [to-draw draw-function]
  [event-type event-function] )
Less Sheldon Cooper, more creation-of-the-universe
big-bang puts all the pieces together to start the world in motion.
initial state
some event happens…
next state
next-state

now the current state
some event happens...
next state

now the current state
some event happens…
next-state

now the current state
Some event happens?
(big-bang initial-state
  [to-draw draw-function]
  [event-type event-function])
(big-bang initial-trafficlight
  [to-draw draw-function]
  [event-type event-function])
(big-bang initial-trafficlight
[to-draw draw-light]
[event-type event-function])
(big-bang initial-trafficlight
to-draw draw-light
[event-type event-function])

We haven’t written this; add it to your wishlist!
(big-bang initial-trafficlight
  [to-draw draw-light]
  [on-tick next-light])
Another function for the wishlist!
So far...

;;; TrafficLight data
;;; - definition
;;; - interpretation
;;; - examples
;;; - template

;;; call big-bang

;;; Wishlist:
;;; (define (draw-light ...))
;;; (define (next-light ...))
Data

- Data definition
- Interpretation
- Examples
- Template

Functions

- Signature
- Purpose
- Tests
- Code
;; draw-light : TrafficLight -> Image
;; draw-light : TrafficLight -> Image

;; next-light : TrafficLight -> TrafficLight
<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data definition</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td><strong>Interpretation</strong></td>
<td><strong>Purpose</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td><strong>Tests</strong></td>
</tr>
<tr>
<td><strong>Template</strong></td>
<td><strong>Code</strong></td>
</tr>
</tbody>
</table>
;; draw-light : TrafficLight -> Image
;; Draws a circle of the given color, rendering a
;; traffic light

;; next-light : TrafficLight -> TrafficLight
;; draw-light : TrafficLight → Image
;; Draws a circle of the given color, rendering a traffic light

;; next-light : TrafficLight → TrafficLight
;; Produces the next light in the sequence TL-GREEN, TL-YELLOW, TL-RED
Data

Data definition
Interpretation
Examples
Template

Functions

Signature
Purpose
Tests
Code
;; draw-light : TrafficLight → Image
;; Draws a circle of the given color, rendering a traffic light
(check-expect (draw-light TL-GREEN)
  (circle 20 "solid" "green"))
(check-expect (draw-light TL-YELLOW)
  (circle 20 "solid" "yellow"))
(check-expect (draw-light TL-RED)
  (circle 20 "solid" "red"))

;; next-light : TrafficLight → TrafficLight
;; Produces the next light in the sequence TL-GREEN, TL-YELLOW, TL-RED
;; draw-light : TrafficLight -> Image
;; Draws a circle of the given color, rendering a traffic light
(check-expect (draw-light TL-GREEN)
  (circle 20 "solid" "green"))
(check-expect (draw-light TL-YELLOW)
  (circle 20 "solid" "yellow"))
(check-expect (draw-light TL-RED)
  (circle 20 "solid" "red"))

;; next-light : TrafficLight -> TrafficLight
;; Produces the next light in the sequence TL-GREEN, TL-YELLOW, TL-RED
(check-expect (next-light TL-GREEN) TL-YELLOW)
(check-expect (next-light TL-YELLOW) TL-RED)
(check-expect (next-light TL-RED) TL-GREEN)
<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data definition</td>
<td>Signature</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Purpose</td>
</tr>
<tr>
<td>Examples</td>
<td>Tests</td>
</tr>
<tr>
<td>Template</td>
<td>Code</td>
</tr>
</tbody>
</table>
[trafficlight.rkt]
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Gregor Kiczales
Shriram Krishnamurthi
Laney Strange