Booleans and if expressions
#true
#false

Can be abbreviated to #t and #f
Primitives or other functions that produce a Boolean value are called *predicates*, e.g., for numbers,

> (> 100 100)  #false

> (= 100 100)  #true

or for strings

> (string=? "foo" "bar")  #false
(require 2htdp/image)

(define I1 (rectangle 10 20 "solid" "red"))

(define I2 (rectangle 20 10 "solid" "blue"))

(< (image-width I1) (image-width I2))
(require 2htdp/image)

(define rect (rectangle 10 20 "solid" "red"))

(if (< (image-width rect) (image-height rect))
  "tall"
  "wide")
To form an **if** expression:

\[
(\text{if } \langle \text{expression} \rangle \langle \text{expression} \rangle \langle \text{expression} \rangle )
\]

- True–false question
- True (“then”) answer
- False (“else”) answer
Evaluation rule for if expressions

1. If the question expression is not a value, evaluate it, and replace with value

2. If the question is true, replace entire if expression with true answer expression

3. If the question is false, replace entire if expression with false answer expression

4. The question is a value other than true or false, so produce an error
To combine Boolean values, we can use **and**:  

\[
\text{(and } \langle expr1 \rangle \langle expr2 \rangle \ldots \text{)}
\]

and **or**:  

\[
\text{(or } \langle expr1 \rangle \langle expr2 \rangle \ldots \text{)}
\]

Evaluation of **and** stops – is “short-circuited” – as soon as an expression evaluates to false.

Evaluation of **or** stops as soon as an expression evaluates to true.
> (and #true #false)  
#false

> (or #true #false)  
#true

> (and (< 1 2) (> 2 3))  
#false

> (or (<= 1 0) (= 1 1))  
#false
To change an expression that evaluates to #true to be #false or vice versa, use not.

> (not (= 1 0))
#true
Aside: Using the Stepper
Function definitions
(require 2htdp/image)

;; Draw a traffic light

(above (circle 40 "solid" "red")
 (circle 40 "solid" "yellow")
 (circle 40 "solid" "green"))

Unchanging  Varying
Remember functions from middle-school math:

Given \( f(x) = 2 \cdot x \)

\( f(2) = 2 \cdot 2 = 4 \)

\( f(6) = 2 \cdot 6 = 12 \)

Parameter stands for varying value
(require 2htdp/image)

;; Draw a traffic light

(above (circle 40 "solid" "red")
  (circle 40 "solid" "yellow")
  (circle 40 "solid" "green"))

;; Can be changed to

(define (bulb color)
  (circle 40 "solid" color))

(above (bulb "red")
  (bulb "yellow")
  (bulb "green"))
To form a function definition:

\[(\text{define} \ (\langle \text{function-name} \rangle \ \langle \text{arg-name} \rangle \ \ldots) \ \langle \text{expression} \rangle)\]
We started by talking about *computation*, e.g., how do we interpret the expression

\[
(+ 1 (* 2 3))
\]

\[
(+ 1 6)
\]

7

But as we start writing function definitions, we need to think about *programming*. 
How to design programs
Following a *design recipe* systematizes the design of a function.

It helps ensure we end up with a high-quality function that does what we need and has been well-tested.
Design recipes will make hard problems much easier – but they also make easy problems a little slower to design.

We’ll use the recipe on easy problems just to learn the recipe.

Hang in there!
Design Recipe 1

1. Data
2. Signature, purpose, and stub
3. Examples
4. Template and inventory
5. Body
6. Test and debug
Problem

Design a function that consumes a number and produces twice that number. Call your function `double`. 
Design a function that consumes a number and produces twice that number. Call your function `double`.
Design a function that consumes a number and produces twice that number. Call your function **double**.

We’re working *directly with numbers*.
Design a function that consumes a number and produces twice that number. Call your function `double`.

**Signature** ;; double : Number -> Number
Design a function that consumes a number and produces twice that number. Call your function `double`.

```plaintext
Signature  ;; double : Number -> Number
Purpose    ;; Produce 2 times the given number
```
Design a function that consumes a number and produces twice that number. Call your function \texttt{double}.

\begin{verbatim}
;; double : Number -> Number
;; Produce 2 times the given number
(define (double n) 0)
\end{verbatim}
Design a function that consumes a number and produces twice that number. Call your function `double`.

;; double : Number -> Number
;; Produce 2 times the given number
(define (double n) 0)

(check-expect (double 3) 6)
(check-expect (double 4.2) 8.4)
Design a function that consumes a number and produces twice that number. Call your function double.

;; double : Number -> Number
;; Produce 2 times the given number
(define (double n) (... n))
(check-expect (double 3) 6)
(check-expect (double 4.2) 8.4)
Design a function that consumes a number and produces twice that number. Call your function `double`.

```scheme
;; double : Number -> Number
;; Produce 2 times the given number
(define (double n) (* 2 n))

(check-expect (double 3) 6)
(check-expect (double 4.2) 8.4)
```
Design a function that consumes a number and produces twice that number. Call your function `double`.

```scheme
;; double : Number -> Number
;; Produce 2 times the given number
(define (double n)
  (* 2 n))
(check-expect (double 3) 6)
(check-expect (double 4.2) 8.4)
Both tests passed!
```
Step 1: Data

Choose a representation suitable for the function’s input data.

- Fahrenheit degrees? Number
- Grocery items? String
- Faces? Image
- Wages? Number

In the case of double, we’re representing numbers as ... numbers.
Step 2: Signature, purpose, and stub

Describe (but don’t write) the function.
Step 2: Signature, purpose, and stub

The function signature declares the type of the function’s input(s) and output.

We write the signature as a comment of the form

```
;; ⟨Function name⟩ : ⟨Type⟩ ... → ⟨Type⟩
```

E.g., a function that consumes a single number and produces a number:

```
;; double : Number → Number
;; cat? : Image → Boolean
```
Step 2: Signature, purpose, and stub

The *purpose statement* is a one-line description of what the function will do, e.g.,

- Converts Fahrenheit degrees \( f \) to Celsius-degrees
- Checks whether \( i \) is an image of a cat

This is given as comment after the signature:

```
;; Produce 2 times the given number
```

If it’s clearer, use the parameters names you give in the stub rather than write “the given whatever”.
Step 2: Signature, purpose, and stub

The *stub* or *function header* is like a piece of scaffolding that we’ll put in place for a short period of time, with

- the correct function name,
- the correct number of parameters,
- a dummy result of the correct type.

E.g.,

```
(define (double n) 0)
```
Step 3: Examples

*Examples* help us understand what the function must do. Use multiple examples to illustrate behavior.

Wrapping the examples in `check-expect` statements means they will also serve as unit tests for the completed function, e.g.,

```
(check-expect (double 3) 6)
(check-expect (double 4.2) 8.4)
```
Step 3: Examples

By running the program at this point, we can see that the tests actually run even though they’ll fail. It’s helpful to see that your tests are well-formed early.
**Note:** Every step of the recipe is designed to help with the steps after it, which is why we follow it in order.

E.g., writing the signature tells us what type of output the function produces, which we need in order to write the stub. The purpose tells us how the output relates to the input so we can write the tests.
Step 4: Templates and inventory

Take inventory, to understand what are the givens and what we need to compute.

We’ll see richer templates later; for now it’s just the parameters, e.g.,

```
(define (double n)
  (... n))
```

For the first few weeks, I recommend that you leave the stub and the template as comments rather than modifying them as you go onto the next steps.
Step 5: Body

How to draw an owl

1. Draw some circles
2. Draw the rest of the fucking owl
Step 5: Body

Code the function body.

Use everything written before to know how to complete the function body.

Sometimes it helps to elaborate the examples to show how the expected value could have been produced.
Step 6: Test and debug

Click Run.

Did the tests work? 🎉

If not, there are three possibilities to consider:

1. You gave the wrong expected answer in one of the test cases.
2. The function definition computes the wrong result – it’s a 🐞!
3. Both the examples and the function definition are wrong.
Design Recipe examples
Example: yell
Problem

Design a function called **yell** that takes as input strings like "Hello" and adds "!" to produce strings like "Hello!".

When we say to “design” a function we mean follow the design recipe:

1. Data
2. Signature, purpose, and stub
3. Examples
4. Template and inventory
5. Body
6. Test and debug

Leave behind commented out versions of the stub and template.
;; We're working directly with strings.

;; yell : String -> String
;; Add "!" to the end of s

;; Stub: (define (yell s) " ")

;; Template:
;; (define (yell s)
;;   (... s))

(define (yell s)
  (string-append s "!"))

(check-expect (yell "Hello") "Hello!")
(check-expect (yell "Bye!") "Bye!")
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Gregor Kiczales
Shriram Krishnamurthi
Marc Smith