CS115 - Module 3 - Booleans, Conditionals, and Symbols

Cameron Morland

Fall 2017

Reminder: if you have not already, ensure you:

- Read *How to Design Programs*, sections 4-5

Booleans (`Bool`)

`<`, `>`, and `=` are new functions, each of which produces a boolean value (`Bool`).

```
(< 4 6)
(> 4 6)
(= 5 7)
```

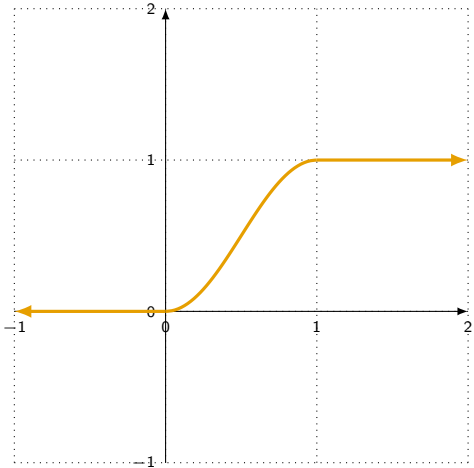
Each produces `#true` or `#false`. These are the only values a `Bool` may take.

(You may see `#true` called `true` or `#t`, and see `#false` called `false` or `#f`.)

Note: keep track of order! `< a b` corresponds to $a < b$.

A function which produces a `Bool` is called a **predicate**. Often the name of predicates end with `?`, as in **`string`**`=?`

Other predicates include `even?` and `odd?`



A sin-squared window, used in signal processing, can be described by

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 1 \\ \sin^2(x\pi/2) & \text{for } 0 \leq x < 1 \end{cases}$$

Racket gives us an easy way to design such things in a special form called **cond**.

Using **cond** to describe cases

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 1 \\ \sin^2(x\pi/2) & \text{for } 0 \leq x < 1 \end{cases}$$

```
(define (ssqw x)
  (cond
    [(< x 0) 0]
    [(>= x 1) 1]
    [(< x 1) (sqr (sin (* x pi 0.5)))]
    ))
```

cond is a special form, not a function. We deal with it in a special way. In particular, *do not evaluate its arguments* until necessary.

Each argument of **cond** is a pair of square brackets around a pair of expressions:

`question answer.`

Evaluating a **cond** statement

How do we evaluate a **cond**?
Informally, evaluate a **cond** by considering the question/answer pairs in order, top to bottom. When considering a question/answer pair, evalute the `question`. If the `question` evaluates to `true`, the *whole* **cond** takes the `answer`.

```
(define (ssqw x)
  (cond
    [(< x 0) 0]
    [(>= x 1) 1]
    [(< x 1) (sqr (sin (* x pi 0.5)))]
    ))
```

For example consider, `(ssqw 4)`.

$\Rightarrow$

```
(cond
   [(< 4 0) 0]
   [(>= 4 1) 1]
   [(< 4 1) (sqr (sin (* 4 pi 0.5)))]
   )
```

Problem solving with **cond**

```
(define (ssqw x)
  (cond
    [(< x 0) 0]
    [(>= x 1) 1]
    [(< x 1) (sqr (sin (* x pi 0.5)))]
    ))
```

Use **cond** to write a function `(absolute-value n)` which produces $|n|$.
(There is a built-in function **abs** which does this, but don't use it now.)

$$a(n) = \begin{cases} -n & \text{if } n < 0 \\ n & \text{if } n \geq 0 \end{cases}$$

```
(define (absolute-value n)
  (cond
   [(< n 0) (- n)]
   [(>= n 0) n]
   ))
```

No satisfied questions

What happens if *none* of the questions evaluate to `true`?

```
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [(< n 0) (- n)]
    ))
```

An error occurs with this `(absolute-value 0)`.

This can be helpful — if we try to consider all the possibilities, but we miss one, testing may raise this error. Then we can fix it.

But sometimes we want to only describe some conditions, and do something different if none of them are satisfied.

We *could* use a question which always evalutes to `true`:

```
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [true (- n)]
    ))
```

Remember: the question/answer pairs are considered *in order*, top to bottom, and it stops as soon as it finds a question which evaluates to `true`.

This is useful sufficiently frequently that there is special keyword for it: `else`.

```
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [else (- n)]
    ))
```

Tracing **cond**

Recall we are imagining interpreting our programs as a series of substitutions, called a **trace**.
How do we trace **cond**?
The general form of a conditional is

```
(cond
 [question1 answer1]
 [question2 answer2]
 ...
 [questionk answerk])
```

To evaluate the conditional, evaluate `question1`, then perform the following substitutions:

- (**cond** [false exp0][exp1 exp2]...) ⇒ (**cond** [exp1 exp2]...)
- (**cond** [true exp0][exp1 exp2]...) ⇒ exp0
- (**cond** [else exp0]) ⇒ exp0

Tracing **cond** example

- (**cond** [false exp0][exp1 exp2]...) ⇒ (**cond** [exp1 exp2]...)
- (**cond** [true exp0][exp1 exp2]...) ⇒ exp0
- (**cond** [else exp0]) ⇒ exp0

```
(define (ssqw x) ...)
(ssqw 0)
⇒ (cond [(< 0 0) 0] [(>= 0 1) 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
⇒ (cond [false 0] [(>= 0 1) 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
⇒ (cond [(>= 0 1) 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
⇒ (cond [false 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
⇒ (cond [(< 0 1) (sqr (sin (* 0 pi 0.5)))])
⇒ (cond [true (sqr (sin (* 0 pi 0.5)))])
⇒ (sqr (sin (* 0 pi 0.5)))
⇒ (sqr (sin 0))
⇒ (sqr 0)
⇒ 0
```

Tracing cond

```
(define (qux a b)
  (cond
    [(= a b) 42]
    [(> a (+ 3 b)) (* a b)]
    [(> a b) (- b a)]
    [else -42]))

(qux 5 4)
```

Perform a complete trace of this program.

Testing `cond`

You should write tests so each `question` is evaluated to `true` at least once, to verify each `answer` is tested.

Include tests for boundaries; it is easy to get "off-by-one" errors!

Suppose I wanted a function which produces 0 for negative numbers, 1 for positive numbers 10 or less, and 2 for other numbers. What should I test?

I should check boundaries (-1, 0, 1) and (10, 11), some other negative number, and some larger number.

`categorize.rkt`

and now for something completely different

We combine predicates using the special forms **and**, `r`, and the function **not**. These all consume and produce `Bool` values.

- **and** produces `false` if at least one of its arguments is `false`, and `true` otherwise.
- **or** produces `true` if at least one of its arguments is `true` and false otherwise.
- **not** produces `true` if its argument is `false`, and `false` if its argument is `true`.

A few examples:

- `(and (> 5 4) (> 7 2))` $\Rightarrow$ `true`
- `(or (> 5 4) (> 7 2))` $\Rightarrow$ `true`
- `(and (> 5 4) (< 7 2))` $\Rightarrow$ `false`
- `(or (> 5 4) (> 7 2))` $\Rightarrow$ `true`
- `(not (= 5 4))` $\Rightarrow$ `true`

An important subtlety interpreting **and** and **or**: short-circuiting

and and **or** are *not* functions. They are **special forms**. Do not evaluate their arguments until necessary.

Informally, evaluate the arguments one by one, and *stop as soon as possible*.

For example:

```
(define (baz x)
  (and (not (= 0 x))
       (> 0 (cos (/ 1 x)))))
```

If I run `(baz 0)`, attempting to evaluate the expression `(/ 1 x)`, would cause a division by zero error. But when `x` is zero, the first argument of **and** is `false`, so the second is not evaluated.

Substitution rules for **and**

Use the following rules for tracing **and**:

- `(and true exp ...)` ⇒ `(and exp ...)`
- `(and false exp ...)` ⇒ `false`
- `(and)` ⇒ `true`

Note: this is not what the stepper does! If in this course you are asked to perform a trace, follow these rules.

Perform a trace of `(and (= 3 3) (> 7 4) (< 7 4) (> 0 (/ 3 0)))`

⇒ `(and true (> 7 4) (< 7 4) (> 0 (/ 3 0)))`
⇒ `(and (> 7 4) (< 7 4) (> 0 (/ 3 0)))`
⇒ `(and true (< 7 4) (> 0 (/ 3 0)))`
⇒ `(and (< 7 4) (> 0 (/ 3 0)))`
⇒ `(and false (> 0 (/ 3 0)))`
⇒ `false`

Substitution rules for `or`

Use the following rules for tracing **`or`**:

- `(or true exp ...)` ⇒ `true`
- `(or false exp ...)` ⇒ `(or exp ...)`
- `(or)` ⇒ `false`

Note: this is not what the stepper does! If in this course you are asked to perform a trace, follow these rules.

Perform a trace of `(or (< 7 4) (= 3 3) (> 7 4) (> 0 (/ 3 0)))`

⇒ `(or false (= 3 3) (> 7 4) (> 0 (/ 3 0)))`
⇒ `(or (= 3 3) (> 7 4) (> 0 (/ 3 0)))`
⇒ `(or true (> 7 4) (> 0 (/ 3 0)))`
⇒ `true`

Nested Conditionals

A museum offers free admission for people who arrive after 5 pm. Otherwise, the cost of admission is based on a person's age: age 10 and under are charged $5 and everyone else is charged $10.

Write a function `admission` which produces the admission cost. It consumes two parameters: a `Bool`, `after5?`, and a positive integer, `age`.

Flattening Nested Conditionals

Sometimes it is desirable to flatten the conditionals.

```
;; admission: Bool Nat
    -> Nat
(define (admission
    after5? age)
  (cond
    [after5? 0]
    [else
     (cond
       [(<= age 10) 5]
       [else 10]
       )
```

↔

```
(define (admission
    after5? age)
  (cond
    [after5? 0]
    [(and
      (not after5?)
      (< age 11)) 5]
    [else 10]))
```

↔

```
(define (admission
    after5? age)
  (cond
    [after5? 0]
    [(< age 11) 5]
    [else 10]))
```

Uses of **cond**

Conditionals can be used like any other expression:

```
(define (add-1-if-even n)
  (+ n
     (cond
      [(even? n) 1]
      [else 0])))


(or (= x 0)
    (cond
     [(positive? x) (> x 100)]
     [else (< x -100)]))
```

Black-box and white-box testing

> *"In science, computing, and engineering, a* **black blox** *is a device. . . which can be viewed in terms of its inputs and outputs, without any knowledge of its internal workings." (Wikipedia)*

Black-box testing refers to testing without reference to how the program works. Black-box tests should be written before you write your code. Your examples are black-box tests.

> *"A white box is a subsystem whose internals can be viewed but usually not altered." (Wikipedia)*

White-box testing should exercise every line of code. Design a test to check both sides of every `question` in every **`cond`**.
These tests are designed after you write your code, by looking at how the code works.

Developing a Predicate

I wish to develop a predicate `cat-start-or-end?`, which consumes a `Str` and determines if the `Str` starts or ends with `"cat"`.

A data type for symbols: `Sym`

A symbol is written as a tick `'` followed by the name of the symbol, which follows the same rules as for identifiers (no spaces, some restrictions on characters).
I can represent the four suits using the four symbols `'diamonds` for diamonds, `'clubs` for clubs, `'hearts` for hearts, and `'spades` for spades.

Operations on `Sym`

A `Sym` is an indivisible, "atomic" value.

The **only** operation that is possible is `symbol=?`, which checks if two symbols are equal.

```
(define trump-suit 'hearts)

(define (trump? suit)
  (symbol=? suit trump-suit))
```

When to use `Sym` and `Str`

When should we use `Sym`?

- Any time you have a fixed set of items, and don't need to manipulate them or order them.
- When you are doing many equality comparisons. `symbol=?` is faster than **`string`**`=?`

When shouldn't we use `Sym`?

- Any time you want to operate on items in any way.
- Any time you want to put items in order.

Problems with the contract

Consider the following function:

```
;; check-divide: Num -> ???
(define (check-divide n)
  (cond [(= 0 n) "undefined"]
        [else (/ 1 n)]))
```

What should the contract be?

It could be a `Num` or a `Str` (specifically, `"undefined"`).

Use `anyof` for situations like this.

```
;; check-divide: Num -> (anyof Num Str)
```

Generalized even

`gen-even?` consumes an integer, a symbol, or a string, and produces `true` if the input is `'even`, `"even"`, or an even integer, and `false` otherwise.

```
(gen-even? v) produce true if v is 'even, "even", or an even integer.
```

Write the contract for `gen-even?`

Design examples and tests for `gen-even?`

Write the body of the function `gen-even?`

Checking the type of a value

There's a problem when we go to write the body. We don't know what type `v` is.
Built-in predicates to the rescue!
`number?`, `integer?`, `symbol?`, and **`string`**`?` each have one parameter, and indicate if the value is a `Num`, `Int`, `Sym`, and `Str` respectively.

Write the body of the function `gen-even?`

Generalized equality checking

The built-in predicate `(equal? a b)` produces `true` if `a`, `b` are the same type, and if they have the same value.
This is very handy if things may not be the same type.

Rewrite `gen-even?` using **`equal?`**

Module summary

Become comfortable using **`cond`** expressions, **`and`**, **`or`**, and **`not`**.
Remember how to test these expressions, and know what black-box and white-box testing are.
Make sure you understand short-circuiting in **`and`** and **`or`**.
Become skillful at tracing code which includes **`cond`**, **`and`**, and **`or`**.
Be able to write programs using `Sym`.
Understand the use of `anyof` and be able to use it in your programs.

Before we begin the next module, please

- Read *How to Design Programs*, sections 6-7, omitting 6.2, 6.6, 6.7, and 7.4.