Simple data

Readings:

- HtDP, sections 4-5

Topics:

- Boolean-valued functions
- Conditional expressions
- Example: computing taxes
- Symbols
- Strings

Boolean-valued functions

A function that tests whether two numbers x and y are equal has two possible **Boolean values**: `true` and `false`.

An example application: `(= x y)`.

This is equivalent to determining whether the mathematical **proposition** "$x = y$" is true or false.

Standard Racket uses `#t` and `#true` where we use `true`, and similarly for `#f`, `#false`, and `false`; these will sometimes show up in basic tests and correctness tests. **You should always use `true` and `false`.**

> Other types of comparisons

In order to determine whether the proposition "$x < y$" is true or false, we can evaluate `(< x y)`.

There are also functions for $>$, $\leq$ (written `<=`) and $\geq$ (written `>=`).

Comparisons are functions which consume two numbers and produce a Boolean value. A sample contract:

```
;; = : Num Num → Bool
```

Note that Boolean is abbreviated in contracts.

> Complex relationships

You may have already learned in Math 135 how propositions can be combined using the connectives AND, OR, NOT.

Racket provides the corresponding **`and`**, **`or`**, and `not`.

These are used to test complex relationships.

Example: the proposition "$3 \leq x < 7$" can be computationally tested by evaluating `(and (<= 3 x) (< x 7))`.

> Some computational differences

The mathematical AND and OR connect two propositions.

In Racket, **`and`** and **`or`** may have more than two arguments.

The special form **`and`** has value `true` exactly when all of its arguments have value `true`.

The special form **`or`** has value `true` exactly when at least one of its arguments has value `true`.

The function `not` has value `true` exactly when its one argument has value `false`.

DrRacket only evaluates as many arguments of **and** and **or** as is necessary to determine the value.

Examples:

```
;; Eliminate easy cases first; might not need to do
;; the much slower computation of prime?
(and (odd? x) (> x 2) (prime? x))

;; Avoid dividing by zero
(and (not (= x 0)) (<= (/ y x) c))
(or (= x 0) (> (/ y x) c))
```

> Predicates

A **predicate** is a function that produces a Boolean result.

Racket provides a number of built-in predicates, such as `even?`, `negative?`, and `zero?`.

We can write our own:

```
(define (between? low high numb)
  (and (< low numb) (< numb high)))

(define (can-vote? age)
  (>= age 18))
```

Predicate names ending with a question mark is a **convention**.

Exercise 1

Figure out how to use each predicate in DrRacket.

Be sure you understand when each produces `true` and when it produces `false`.

```
1 >
2 even?
3 string>=?
4 =
5 equal?
```

Conditional expressions

Sometimes, expressions should take one value under some conditions, and other values under other conditions.

Example: taking the absolute value of x.

$$|x| = \begin{cases} -x & \text{when } x < 0 \\ x & \text{when } x \geq 0 \end{cases}$$

In Racket, we can compute $|x|$ with the **conditional expression**

```
(cond [(< x 0)   (- x)]
      [(>= x 0)      x])
```

- Conditional expressions use the special form `cond`.
- Each argument is a question/answer pair.
- The **question** is a Boolean expression.
- The **answer** is a possible value of the conditional expression.
- Square brackets are used by convention, for readability.
- Square brackets and parentheses are equivalent in the teaching languages (must be nested properly).
- `abs` is a built-in function in Racket.

The general form of a conditional expression is

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

where `questionk` could be **`else`**

- The questions are evaluated in top-to-bottom order
- As soon as one question is found that evaluates to `true`, no further questions are evaluated.
- Only one answer is ever evaluated. (the one associated with the first question that evaluates to `true`, or associated with the **`else`** if that is present and reached)
- An error is produced if no question evaluates to `true`.

> Example

$$f(x) = \begin{cases} 0 & \text{when } x = 0 \\ x \sin(1/x) & \text{when } x \neq 0 \end{cases}$$

```
(define (f x)
  (cond [(= x 0) 0]
        [else (* x (sin (/ 1 x)))]))
```

> Simplifying conditional functions

Sometimes a question can be simplified by knowing that if it is asked, all previous questions have evaluated to `false`.

Here are the common recommendations on which course to take after CS 135, based on the mark earned.

- $0\% \leq$ mark $< 40\%$: CS 115 is recommended
- $40\% \leq$ mark $< 50\%$: CS 135 is recommended
- $50\% \leq$ mark $< 60\%$: CS 116 is recommended
- $60\% \leq$ mark: CS 136 is recommended

We might write the tests for the four intervals this way:

```
(define CS115 1)
(define CS116 2)
(define CS135 3)
(define CS136 4)

(define (course-after-cs135 grade)
  (cond [(< grade 40) CS115]
        [(and (>= grade 40) (< grade 50)) CS135]
        [(and (>= grade 50) (< grade 60)) CS116]
        [(>= grade 60) CS136]))
```

We can simplify three of the tests.

```
(define CS115 1)
(define CS116 2)
(define CS135 3)
(define CS136 4)

(define (course-after-cs135 grade)
  (cond [(< grade 40) CS115]
        [(< grade 50) CS135]
        [(< grade 60) CS116]
        [else CS136]))
```

These simplifications become second nature with practice.

Exercise 2

Simplify.

```
;; (flatten-me x) Say which interval x is in.
;; flatten-me:  Nat → Str

(define (flatten-me x)
  (cond [(< x 25) "first"]
        [(and (>= x 25) (< x 50)) "second"]
        [(and (>= x 50) (< x 75)) "third"]
        [(>= x 75) "fourth"]))
```

Tests for conditional expressions

- Write at least one test for each possible answer in the expression.
- That test should be simple and direct, aimed at testing that answer.
- When the problem contains **boundary conditions** (like the cut-off between passing and failing), they should be tested explicitly.
- DrRacket highlights unused code.

For the example above:

```
(define CS115 1)
(define CS116 2)
(define CS135 3)
(define CS136 4)


(define (course-after-cs135 grade)
  (cond [(< grade 40) CS115]
        [(< grade 50) CS135]
        [(< grade 60) CS116]
        [else CS136]))
```

there are four intervals and three boundary points, so seven tests are required (for instance, 30, 40, 45 50, 55, 60, 70).

Testing **`and`** and **`or`** expressions is similar.

For `(and (not (zero? x)) (<= (/ y x) c))`, we need:

- one test case where x is zero
  (first argument to **`and`** is `false`)
- one test case where x is nonzero and $y/x > c$,
  (first argument is `true` but second argument is `false`)
- one test case where x is nonzero and $y/x \leq c$.
  (both arguments are `true`)

Some of your tests, including your examples, will have been defined before the body of the function was written.

These are known as **black-box tests**, because they are not based on details of the code.

Other tests may depend on the code, for example, to check specific answers in conditional expressions.

These are known as **white-box tests**. Both types of tests are important.

Exercise 3

Write a function that consumes a `Num`, x, and produces

- `"big"` if $80 < x \leq 100$,
- `"small"` if $0 < x \leq 80$,
- `"invalid"` otherwise.

Write tests to verify the boundaries are where they should be.

> Writing Boolean tests

The textbook writes tests in this fashion:

```
(= (sum-of-squares 3 4) 25)
```

which works outside the teaching languages.

`check-expect` was added to the teaching languages after the textbook was written. You should use it for all tests.

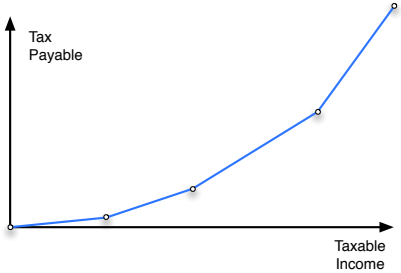
Example: computing taxes

Purpose: Compute the Canadian tax payable on a specified income.

Background:

- 15% on the amount in [$0, $45,000]
- 20% on the amount in ($45,000, $90,000]
- 25% on the amount in ($90,000, $150,000]
- 30% on the amount in ($150,000, $200,000]
- 35% on the amount over $200,000

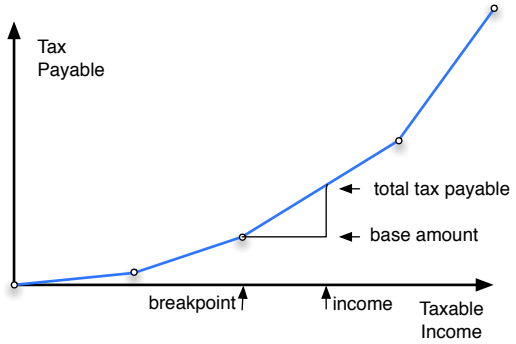
Note: These amounts are rounded from the actual amounts to make discussing them easier.



The “piecewise linear” nature of the graph complicates the computation of tax payable.

One way to do it uses the **breakpoints** (x-value or salary when the rate changes) and **base amounts** (y-value or tax payable at breakpoints).

This is what the paper Canadian tax form does.



> Examples:

Income	Tax Calculation
\$40,000	$0.15 * 40000 = 6000$
\$60,000	$0.15 * 45000 + 0.20 * (60000 - 45000) = 6750 + 3000 = 9750$
\$100,000	$0.15 * 45000 + 0.20 * (90000 - 45000) + 0.25 * (100000 - 90000)$ $= 6750 + 9000 + 2500 = 18250$

```
(check-expect (tax-payable 40000) (* 0.15 40000))
(check-expect (tax-payable 60000) (+ (* 0.15 45000)
                                     (* 0.20 (- 60000 45000))))
(check-expect (tax-payable 100000) 18250)
```

Definition header & contract

```
;; tax-payable: Num → \ Num
;;    requires: income ≥ 0

(define (tax-payable income) ...)
```

Finalize purpose

```
;; (tax-payable income) computes the Canadian tax payable on income.
```

> Write function body

Some constants will be useful. Put these before the purpose and other design recipe elements.

```
;; Rates
(define rate1 0.15)
(define rate2 0.20)
(define rate3 0.25)
(define rate4 0.30)
(define rate5 0.35)
```

```
;; Breakpoints in increasing order by income
(define bp1 45000)
(define bp2 90000)
(define bp3 150000)
(define bp4 200000)
```

Instead of putting the base amounts into the program as numbers (as the tax form does), we can compute them from the breakpoints and rates.

```
;; Base Amounts
;; basei is the base amount for interval [bpi,bp(i+1)]
;; that is, tax payable at income bpi
(define base1 (* (- bp1 0) rate1))
(define base2 (+ base1 (* rate2 (- bp2 bp1))))
(define base3 (+ base2 (* rate3 (- bp3 bp2))))
(define base4 (+ base3 (* rate4 (- bp4 bp3))))
```

```
;; tax-payable: Num → Num
;;   requires:  income >= 0.00
(define (tax-payable income)
  (cond [(< income bp1) (* rate1 income)]
        [(< income bp2) (+ base1 (* rate2 (- income bp1)))]
        [(< income bp3) (+ base2 (* rate3 (- income bp2)))]
        [(< income bp4) (+ base3 (* rate4 (- income bp3)))]
        [else (+ base4 (* rate5 (- income bp4)))]))
```

There are many similar calculations in the tax program, leading to the definition of the following **helper function**:

```
;; (cum-tax base rate low high) calculates the cumulative tax owed
;; where base is the tax owed on income up to low and rate is
;; the tax rate on income between low and high.
;; cum-tax: Num Num Num Num → Num
;;   requires base >= 0, rate >=0, 0 <= low <= high
(define (cum-tax base rate low high)
  (+ base (* rate (- high low))))
```

It can be used for defining constants and the main function.

`tax-payable` with a helper

```
;; Base Amounts
(define base1 (cum-tax 0 rate1 0 bp1))
(define base2 (cum-tax base1 rate2 bp1 bp2))
(define base3 (cum-tax base2 rate3 bp2 bp3))
(define base4 (cum-tax base3 rate4 bp3 bp4))


(define (tax-payable income)
  (cond [(< income bp1) (cum-tax 0 rate1 0 income)]
        [(< income bp2) (cum-tax base1 rate2 bp1 income)]
        [(< income bp3) (cum-tax base2 rate3 bp2 income)]
        [(< income bp4) (cum-tax base3 rate4 bp3 income)]
        [else (cum-tax base4 rate5 bp4 income)]))
```

See HtDP, section 3.1, for a good example of helper functions.

Helper functions are used for three purposes:

- Reduce repeated code by generalizing similar expressions.
- Factor out complex calculations.
- Give names to operations.

Style guidelines:

- Improve clarity with short definitions using well-chosen names.
- Name all functions (including helpers) meaningfully; not “helper”.
- Purpose, contract, and one example are required.

Symbolic data

In earlier examples we used the following constants:

```
(define CS115 1)
(define CS116 2)
(define CS135 3)
(define CS136 4)
```

Note that we didn't actually perform any computations with `CS115`; it was just a way to indicate which course to take next.

Racket allows one to define and use **symbols** with meaning to us (not to Racket).

A symbol is defined using an apostrophe or 'quote': `'CS115`

`'CS115` is a value just like `0` or `115`, but it is more limited computationally.

Symbols can be compared using the predicate `symbol=?`.

```
(define home 'Earth)
```

`(symbol=? home 'Mars)` ⇒ `false`

`symbol=?` is the only function we'll use in CS135 that is applied only to symbols.

Unlike numbers, symbols are self-documenting – you don't need to define constants for them.

Exercise 4

First create a constant: `(define mysymbol 'blue)`

Then see what each of these expressions evaluates to:

```
(equal? mysymbol 42)
(equal? mysymbol "blue")
(equal? mysymbol 'blue)
(equal? mysymbol 'red)
(symbol? '*@)
(symbol? "the artist formerly known as Prince")
```

Characters

A **character** is most commonly a printed letter, digit, or puctuation symbol. *a*, *G*, ., $+$, and 8 are all characters.

Other characters represent less visible things like a tab or a newline in text.

More recent characters include ☺, 🖂, and ☰.

For now, we'll be interested in characters only because they are the simplest component of a **string**.

Strings

Strings are sequences of characters between double quotes. Examples: `"blue"` and `"These are not my shoes. My shoes are brown."`.

What are the differences between strings and symbols?

- Strings are really **compound data**
  (a string is a sequence of characters).
- Symbols can't have certain characters in them
  (such as spaces).
- More efficient to compare two symbols than two strings
- More built-in functions for strings

Here are a few functions which operate on strings.

```
(string-equal? "alpha" "bet") ⇒ false
(string<? "alpha" "bet") ⇒ true
(string-append "alpha" "bet") ⇒ "alphabet"
(string-length "perpetual") ⇒ 9
(string-upcase "Hello") ⇒ "HELLO"
```

The textbook does not use strings; it uses symbols.

We will be using both strings and symbols, as appropriate.

Symbols vs. strings

Consider the use of symbols when a small, fixed number of labels are needed (e.g. planets) and comparing labels for equality is all that is needed.

Use strings when the set of values is more indeterminate (e.g. names of students), or when more computation is needed (e.g. comparison in alphabetical order).

When these types appear in contracts, they should be capitalized and abbreviated: `Sym` and `Str`.

General equality testing

Every type seen so far has an equality predicate
(e.g, `=` for numbers, `symbol=?` for symbols, `string=?` for strings).

The predicate `equal?` can be used to test the equality of two values which may or may not be of the same type.

```
(= 10 11) ⇒ false
(string-equal? "10" "10") ⇒ true
(= 10 "10") ⇒ Error
(equal? 10 "10") ⇒ false
```

`equal?` works for all types of data we have encountered so far (except inexact numbers), and most types we will encounter in the future.

Do not overuse `equal?`.

If you know that your code will be comparing two numbers, use `=` instead of `equal?`.

Similarly, use `symbol=?` if you know you will be comparing two symbols.

This gives additional information to the reader, and helps catch errors (if, for example, something you thought was a symbol turns out not to be one).

Goals of this module

- You should understand Boolean data, and be able to perform and combine comparisons to test complex conditions on numbers.
- You should understand the syntax and use of a conditional expression.
- You should understand how to write `check-expect` examples and tests, and use them in your assignment submissions.
- You should be aware of other types of data (symbols and strings), which will be used in future lectures.
- You should look for opportunities to use helper functions to structure your programs, and gradually learn when and where they are appropriate.

Exercise 5

Use `string-append` and `substring` to complete the function `chop-word`:

```
;; (chop-word s): select some pieces of s.
;; chop-word: Str → Str
;; Examples:
(check-expect (chop-word "In a hole in the ground there lived a hobbit.")
              ;;          ^    ^    ^    ^    ^    ^    ^    ^    ^
              ;;  index:  0    5   10   15   20   25   30   35   40
              "a hobbit lived in the ground")
(check-expect (chop-word "In a town by the forest there lived a rabbit.")
              ;;          ^    ^    ^    ^    ^    ^    ^    ^    ^
              ;;  index:  0    5   10   15   20   25   30   35   40
              "a rabbit lived by the forest")
(check-expect (chop-word "ab c defg hi jkl mnopqr stuvw xyzAB C DEFGHIJ")
              "C DEFGHI xyzAB hi jkl mnopqr")
```

Exercise 6

Use the constants `the-str` and `len-str`, along with the string functions `string-append`, `string-length`, and `number->string` to complete the function `describe-string`:

```
(define the-str "The string '")
(define len-str "' has length ")



;; (describe-string s) Say a few words about s.
;; describe-string: Str → Str
;; Examples:
(check-expect (describe-string "foo") "The string 'foo' has length 3")
(check-expect (describe-string "") "The string '' has length 0")
```