Simple data

Readings:

- HtDP, sections 4-5

Topics:

- Boolean-valued functions
- Symbolic data
- Strings
- Conditional expressions
- Example: computing taxes

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, 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, OR connect two propositions.

The Racket 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 time-consuming factorization in 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))
```

Symbolic data

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

A symbol is defined using an apostrophe or ‘quote’: ’Earth

’Earth is a value just like 6, but it is more limited computationally.

It allows a programmer to avoid using constants to represent names of colours, or of planets, or of types of music.

```
(define Mercury 1) (define Venus 2) (define Earth 3)
```

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

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.

Strings

Racket also supports strings, such as `"blue"`.

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-append "alpha" "bet") ⇒ "alphabet"
(string-length "perpetual") ⇒ 9
(string<? "alpha" "bet") ⇒ true
```

The textbook does not use strings; it uses symbols.

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

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

Use strings when the set of values is more indeterminate, 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.

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

However, 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).

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 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 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)

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 (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 (course-after-cs135 grade)
  (cond [(< grade 40) 'cs115]
        [(< grade 50) 'cs135]
        [(< grade 60) 'cs116]
        [else 'cs136]))
```

These simplifications become second nature with practice.

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 (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.

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.

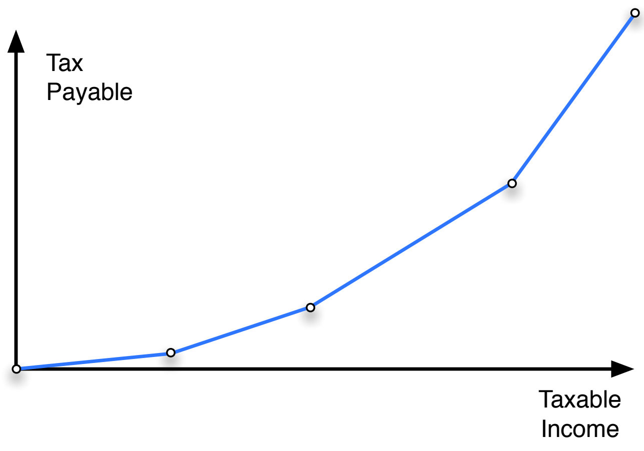
Example: computing taxes

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

Examples:

Google “Canada income tax” For 2017:

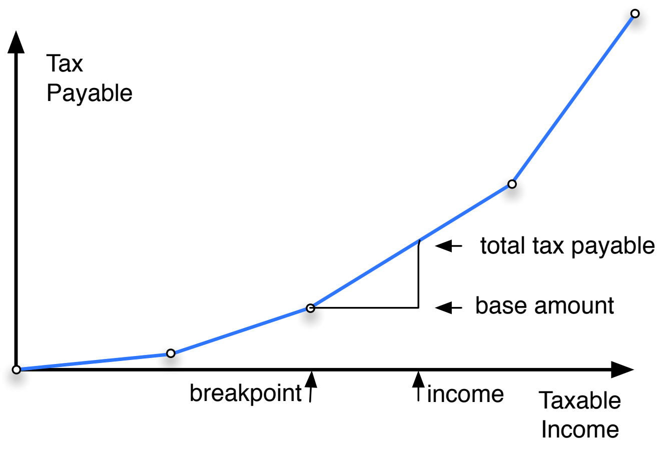
- 15% on the amount in [$0 to $45,916]
- 20.5% on the amount in ($45,916 to $91,831]
- 26% on the amount in ($91,831 to $142,353]
- 29% on the amount in ($142,353 to $202,800]
- 33% on the amount over $202,800



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
\$45,000	$0.15 * 45000 = 6750$
\$50,000	$0.15 * 45916 + 0.205 * (50000 - 45916)$
	$= 7724.62$
\$100,000	$0.15 * 45916 + 0.205 * (91831 - 45916)$
	$+0.26 * (100000 - 91831) = 18423.915$

```
(check-expect (tax-payable 45000) 6750)
(check-expect (tax-payable 50000) 7724.62)
(check-expect (tax-payable 100000) 18423.915)
```

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 0.205)
(define rate3 0.26)
(define rate4 0.29)
(define rate5 0.33)
```

```
;; Breakpoints
(define bp1 45916)
(define bp2 91831)
(define bp3 142353)
(define bp4 202800)
```

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.

```
;; 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 (* (− bp2 bp1) rate2)))
(define base3 (+ base2 (* (− bp3 bp2) rate3)))
(define base4 (+ base3 (* (− bp4 bp3) rate4)))
```

```
;; tax-payable: Num → Num
;; requires: income ≥ 0
(define (tax-payable income)
  (cond [(< income 0) 0]      ;; Not strictly necessary given contract
        [(< income bp1) (* income rate1)]
        [(< income bp2) (+ base1 (* (− income bp1) rate2))]
        [(< income bp3) (+ base2 (* (− income bp2) rate3))]
        [(< income bp4) (+ base3 (* (− income bp3) rate4))]
        [else (+ base4 (* (− income bp4) rate5))]))
```

Helper functions

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

```
;; (tax-calc base rate low high) calculates the total tax owed ...
;; tax-calc: Num Num Num Num → Num
;; requires base ≥ 0, rate ≥ 0, 0 ≤ low ≤ high
;; Example:
(check-expect (tax-calc 1000 0.10 10000 10100) 1010)
(define (tax-calc base rate low high) (+ base (* rate (− high low))))
```

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

```
(define base1 (tax-calc 0 rate1 0 bp1))
(define base2 (tax-calc base1 rate2 bp1 bp2))
(define base3 (tax-calc base2 rate3 bp2 bp3))
(define base4 (tax-calc base3 rate4 bp3 bp4))

(define (tax-payable income)
  (cond [(< income 0) 0]   ; Not strictly necessary
        [(< income bp1) (tax-calc 0 rate1 0 income)]
        [(< income bp2) (tax-calc base1 rate2 bp1 income)]
        [(< income bp3) (tax-calc base2 rate3 bp2 income)]
        [(< income bp4) (tax-calc base3 rate4 bp3 income)]
        [else (tax-calc 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.

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.