Module 4: 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

\[(\text{and} \ (\leq 3 \ x) \ (\lt x \ 7)).\]
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*.
Some computational differences

Truth tables for **and**, **or** and **not**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>(and p q)</th>
<th>(or p q)</th>
<th>(not q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>
Some computational differences

Some observations about \texttt{and} and \texttt{or}

- In some cases you do not need to evaluate every argument to know the answer

\begin{align*}
\texttt{(and false x)} &= \text{false} & \text{but} \quad \texttt{(and true x)} &= x \\
\texttt{(or true x)} &= \text{true} & \text{but} \quad \texttt{(or false x)} &= x
\end{align*}

- This observation leads to the idea of \underline{short-circuiting}, i.e. only evaluating the arguments until you know the answer.

- Because not all arguments are evaluated, \texttt{and} and \texttt{or} are special forms.
Short-circuiting

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.

Symbols allow 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.
Comparing Symbols

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.
Symbols

- Symbols are used when you want to classify values into a few categories: e.g.
  - e.g. food: ’Chinese, ’Mexican, ’Thai, ’Italian
  - e.g. courses: ’interesting, ’ok, ’boring
  - e.g. movies: ’excellent, ’good, ’average, ’bad

- You can test to see if a result is a certain symbol using (symbol=? )

- The contract is symbol=?: Sym → Bool
Strings

Racket also supports strings, such as "blue".

What are the differences between strings and symbols?

- Strings are really *compound data* (i.e. 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
Strings

• A **string** is a sequence of characters, i.e. commonly called text.

• There are many move functions available for strings (compared to symbols). Strings can be
  – compared sorted alphabetically with `string<`
  – have a length, i.e. `string-length`
  – and can be joined together, i.e. `string-append`.

• Symbols are more like multiple choice questions (a few predictable options) and strings are like essay answers (many possible options).
String Functions

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.
Strings vs. Symbols

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 almost all types of data* we have encountered so far (except inexact numbers), and most types we will encounter in the future.
When to Avoid equal?

_Do not overuse equal?_ ⇒ use a more specific predicate if the types are both known to be the same.

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).