Module 3: Making decisions

Readings: Sections 4 and 5 of HtDP.

A Racket program applies functions to values to compute new values. These new values may in turn be supplied as arguments to other functions.

So far, we have seen values that are numbers and strings.

In this module, we will introduce a new type of value.

Throughout the course, we will learn how to construct new types of values.

The data type Boolean (Bool)

A **Boolean value** is either true or false.

A **Boolean function** produces a Boolean value.

Racket provides many built-in Boolean functions (for example, to do comparisons).

(= x y) is equivalent to determining whether "$x = y$" is true or false, for numbers x and y.

```
;; = : Num Num → Bool
```

Boolean values in DrRacket

- Additional constants: #true is equivalent to true, and #false is equivalent to false.
- Within DrRacket, you can choose to use either set of values. DrRacket uses #true and #false by default. We will use true and false in our notes.
- To switch from the default, when setting your language level in DrRacket, choose ”Show Details”, and choose your preferred set of constants.
- There are also additional constants: #t for true and #f for false.

Other types of comparisons

A **comparison** is a function that consumes two numbers and produces a Boolean value.

Other comparisons:

```
(< x y)
(> x y)
(<= x y)
(>= x y)
```

We can also compare strings using `string=?`, `string<?`, and so on.

Complex relationships

You may have learned in a math class how mathematical statements can be combined using the connectives AND, OR, NOT.

Racket provides the corresponding and, or, not.

These are used to check complex relationships.

The statement "$3 \leq x < 7$" is the same as "$x \in [3, 7)$", and can be checked by evaluating

(and (<= 3 x) (< x 7)).

Some computational differences

The special forms and and or can each have two or more arguments.

The special form and simplifies to true exactly when all of its arguments have value true.

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

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

The arguments of and and or are evaluated in order from left to right.

The evaluation stops as soon as the value can be determined. (This is called *short circuit evaluation*.)

Not all arguments may be evaluated. (This is why and and or are called *special forms* rather than *functions*.)

```
(and (>= (string-length str) 3)
     (string=? "cat" (substring str 0 3)))
(or (= x 0) (> (/ 100 x) 5))
```

Substitution rules for and

Here are the simplifications for application of and (slightly different from the textbook).

(and true exp . . .) $\Rightarrow$ (and exp . . .).

(and false exp . . .) $\Rightarrow$ false.

(and) $\Rightarrow$ true.

The last rule is needed when all arguments evaluate to true.

```
(define (good? str)
  (and (>= (string-length str) 3)
       (string=? "cat" (substring str 0 3))))
(good? "at")
⇒ (and (>= (string-length "at") 3)
      (string=? "cat" (substring "at" 0 3))))
⇒ (and (>= 2 3)
      (string=? "cat" (substring "at" 0 3)))
⇒ (and false (string=? "cat" (substring "at" 0 3)))
⇒ false
```

```
(define str "catch")
(and (>= (string-length str) 3)
     (string=? "cat" (substring str 0 3)))
⇒ (and (>= (string-length "catch") 3)
     (string=? "cat" (substring str 0 3)))
⇒ (and (>= 5 3)
     (string=? "cat" (substring str 0 3)))
⇒ (and true (string=? "cat" (substring str 0 3)))
⇒ (and (string=? "cat" (substring str 0 3)))
⇒ (and (string=? "cat" (substring "catch" 0 3)))
⇒ (and (string=? "cat" "cat")) ⇒ (and true) ⇒ (and) ⇒ true
```

Substitution rules for or

The rules for or are similar, but with the roles of true and false exchanged.

(or true exp . . .) $\Rightarrow$ true.

(or false exp . . .) $\Rightarrow$ (or exp . . .).

(or) $\Rightarrow$ false.

```
(define x 10)

(or (= x 0) (> (/ 100 x) 5))

⇒ (or (= 10 0) (> (/ 100 x) 5))

⇒ (or false (> (/ 100 x) 5))

⇒ (or (> (/ 100 x) 5))

⇒ (or (> (/ 100 10) 5))

⇒ (or (> 10 5))

⇒ (or true)

⇒ true
```

```
(define x 0)
(or (= x 0) (> (/ 100 x) 5))
⇒ (or (= 0 0) (> (/ 100 x) 5))
⇒ (or true (> (/ 100 x) 5))
⇒ true
```

An example trace using and and or

```
(and (< 3 5) (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))
⇒ (and true (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))
⇒ (and (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))
⇒ (and (or true (= 1 3)) (or (> 1 5) (> 2 5)))
⇒ (and true (or (> 1 5) (> 2 5)))
⇒ (and (or (> 1 5) (> 2 5)))
⇒ (and (or false (> 2 5)))
⇒ (and (or (> 2 5)))
⇒ (and (or false))
⇒ (and (or)) ⇒ (and false) ⇒ false
```

Predicates

A **predicate** is a function that produces a Boolean result: true if data is of a particular form, and false otherwise.

Built-in predicates: e.g. even?, negative?, zero?, string?

User-defined (require full design recipe!):

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

(define (can-drink? age)
  (>= age 19))
```

Conditional expressions

Sometimes expressions should take different values under different conditions.

- These use the special form cond.
- Each argument of cond is a question/answer pair.
- The question is a Boolean expression.
- The answer is a possible value of the conditional expression.

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])
```

- 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 Racket function

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 order; as soon as one evaluates to true, the corresponding answer is evaluated and becomes the value of the whole expression.

- 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, that is
  – the one associated with the first question that evaluates to true, or
  – the one associated with the else if that is present and reached (all previous questions evaluate to false).

Substitution rules for cond

There are three substitution rules: when the first expression is false, when it is true, and when it is else.

(cond [false . . .][exp1 exp2]. . .)

⇒ (cond [exp1 exp2]. . .).

(cond [true exp]. . .) ⇒ exp.

(cond [else exp]) ⇒ exp.

Example:

```
(define n 5)
(cond [(even? n) "even"] [(odd? n) "odd"])
⇒ (cond [(even? 5) "even"] [(odd? n) "odd"])
⇒ (cond [false "even"] [(odd? n) "odd"])
⇒ (cond [(odd? n) "odd"])
⇒ (cond [(odd? 5) "odd"])
⇒ (cond [true "odd"])
⇒ "odd"
```

```
(define (safe-reciprocal n)
  (cond [(zero? n) "Undefined"] [else (/ 1 n)]))
(safe-reciprocal 0)
⇒ (cond [(zero? 0) "Undefined"] [else (/ 1 0)])
⇒ (cond [true "Undefined"] [else (/ 1 0)])
⇒ "Undefined"
```

Nested conditionals

A museum offers free admission for people who arrive after 5pm. 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 that consumes two parameters: a Boolean value, after5?, and positive integer, age, and produces the associated admission cost.

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

Flattening a nested conditional

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

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

Conditional expressions can be used like any other expressions:

```
(define (add1-if-even n)
  (+ n
     (cond
       [(even? n) 1]
       [else 0])))

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

Design recipe modifications

When we add to the language, we adjust the design recipe.

We add new steps and modify old steps.

If we don't mention a step, it is because it has not changed. It does not mean that it is no longer needed!

New step: data analysis

Modified steps: examples, tests

Design recipe modifications

Data analysis: figure out the different cases (possible outcomes).

Determine the inputs that lead to each case. *(This is a ”thinking” step. There is nothing to write up.)*

Examples/Tests: check interiors of intervals and endpoints.

Definition:

- Choose an ordering of the cases.
- Determine questions to distinguish between cases.
- Develop each question-answer pair one at a time.

Revisiting charges-for

Data analysis:

- outcomes: zero charge or charged per minute over free limit
- intervals: minutes below or above free limit
- question: compare minutes to free limit

Examples/Tests:

- minutes below free limit
- minutes above free limit
- minutes equal to free limit

```
(define (charges-for minutes freelimit rate)
  (cond
    [(< minutes freelimit) 0]
    [else (* (- minutes freelimit) rate)]))

(define (cell-bill day eve)
 (+ (charges-for day day-free day-rate)
    (charges-for eve eve-free eve-rate)))
```

Bonus mark example

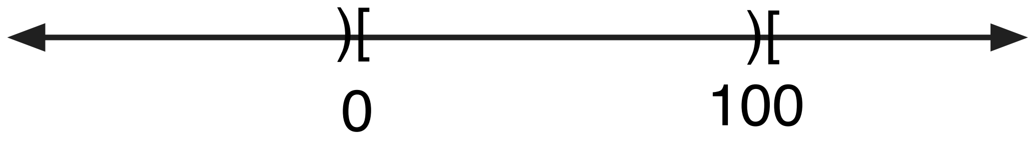
bonus determines bonus marks based on the difference between the sum of the first three assignments and the sum of the last three assignments.

bonus consumes the difference and produces the bonus:

- zero if the difference was negative
- the difference itself for an increase of up to 100
- double the difference for an increase of 100 or more

For a difference diff, the cases for a bonus are 0, diff, and $(* \; 2 \; diff)$.

Here are the intervals for each case:



Examples/Tests: -50, 0, 50, 100, 125

Useful constants:

```
(define no-bonus 0)
(define single-bonus 100)
```

```
;; bonus: Num → Num
(define (bonus diff)
  (cond [(< diff no-bonus) 0]
        [(< diff single-bonus) diff]
        [else (* 2 diff)]))
```

Tests for conditional expressions

- Write *at least one* test for each possible answer.
  - When combination Boolean expressions are used, more tests may be needed (*more later...*).
- The test should be simple and directed to a specific answer.
- The purpose of each test should be clear.
- When the problem contains boundary conditions (like a cut-off between passing and failing), they should be tested explicitly.
- Note that DrRacket highlights unused code.

Testing bonus marks

```
(define (bonus diff)
  (cond
    [(< diff no-bonus) 0]
    [(< diff single-bonus) diff]
    [else (* 2 diff)]))
;; Tests for bonus
(check-expect (bonus −50) 0)   ; first interval
(check-expect (bonus 0) 0)     ; boundary
(check-expect (bonus 50) 50)   ; second interval
(check-expect (bonus 100) 200) ; boundary
(check-expect (bonus 125) 250) ; third interval
```

Testing Boolean expressions

For and: one test case that produces true and one test case for each way of producing false.

For or: one test case that produces false and one test case for each way of producing true.

```
(and (>= (string-length str) 3)
     (string=? "cat" (substring str 0 3)))
```

"catch" (length at least three, starts with "cat"),
"at" (length less than three),
"dine" (length at least three, doesn't start with "cat").

Black-box tests and white-box tests

Some tests may be chosen before the function is written, based on what the function is supposed to do. These are called **black-box tests**. Examples should be chosen from black-box tests.

Other tests may be chosen to exercise different parts of the code. These are called **white-box tests**, and include tests for

- at least each line of code (including thorough testing of Boolean expressions), and
- each way that a question of a cond can be made true.

Use both: black-box before coding, white-box after coding.

Boolean tests

The textbook writes tests in this fashion:

```
(= (bonus -50) 0)
```

You can visually check these tests by looking for trues.

These tests work outside of the teaching languages.

We will continue to use check-expect and check-within as previously described.

Substring checking

cat-start-or-end? asks if a string starts or ends with `"cat"`.

Possible outcomes: true and false.

Inputs leading to true: starts with `"cat"`, ends with `"cat"`, starts and ends with `"cat"`.

Questions to ask:

Is the string too short to contain `"cat"`?

Does the string start with `"cat"`?

Does the string end with `"cat"`?

Developing predicates

```
(define (too-short? s)
  (> 3 (string-length s)))

(define (cat-start? s)
  (string=? "cat" (substring s 0 3)))

(define (cat-end? s)
  (string=? "cat" (substring s (- (string-length s) 3) )))
```

Refined data analysis for cat-start-or-end?:

- Too short to contain `"cat"`: produces false.
- Starts with `"cat"`: produces true.
- Ends with `"cat"`: produces true.
- Long enough but doesn't start or end with `"cat"`: produces false.

Examples: `"me"`, `"caterpillar"`, `"polecat"`, and `"no cat here"`.

```
(define (cat-start-or-end? s)
  (cond
    [(too-short? s) false]
    [(cat-start? s) true]
    [(cat-end? s) true]
    [else false]))
;; Tests for cat-start-or-end?
(check-expect (cat-start-or-end? "me") false)
(check-expect (cat-start-or-end? "caterpillar") true)
(check-expect (cat-start-or-end? "polecat") true)
(check-expect (cat-start-or-end? "no cat here") false)
```

Mixed data

Sometimes a function will consume one of several types of data, or will produce one of several types of data. Sometimes, a function will produce only a small number of values of a specific type.

```
;; safe-reciprocal: Num → (anyof ”Undefined” Num)
(define (safe-reciprocal n)
  (cond [(zero? n) "Undefined"]
        [else (/ 1 n)]))
```

The produced type could also be given as (anyof Str Num), but the above form is more specific.

More on mixed data

If we will be using the result of safe-reciprocal in a program, it may be helpful to assign a new type name to this extended number type. For example,

```
;; An ExtNum is one of
;; * Num
;; * "Undefined"
```

This is called a **data definition**.

Data definitions

Data definitions are made up of at most three parts:

- the first part provides type information about the type being defined. This part is mandatory.
- a descriptive ”where” section, which provides additional, non-type information about the components of the data, often describing their roles. This is included only if needed.
- a technical ”requires” section, whic provides additional conditions not covered by the type information. This is included only if needed.

Building on the ExtNum data definition

We can use ExtNum in contracts as a consumed or produced type.

The data definition will also be helpful as we write functions that consume values of type ExtNum, as the structure of our functions often mirror the shape of our data. This observation is the basis for **data-directed design**, which we will follow as we introduce new types throughout the rest of the course.

For example, any function that consumes an ExtNum value likely needs to check whether the consumed value is a number or a string before processing it.

Racket includes a **type predicate** called number? that consumes any type of value and produces true if the value is a number and false otherwise.

So, all functions that consume an ExtNum value will have the same basic structure, which we call a **template** for such functions:

```
;; extnum-template: ExtNum → Any
(define (extnum-template x)
          (cond [(number? x) . . . ]
                [else . . . ]))
```

For example,

```
;; ext-neg: ExtNum → ExtNum
(define (ext-neg x)
    (cond [(number? x) (− x)]
          [else "Undefined"]))
```

More formally

If the consumed or produced data can be any of the types t1, t2, ..., tN, we write:

(anyof t1 t2 . . . tN)

If the consumed or produced data can be any of the types t1, t2, ..., tN, or any of the values v1, v2, ..., vT, we write:

(anyof t1 t2 . . . tN v1 v2 . . . vT)

If a function can consume or produce different types of data, you will need to make a decision to determine how to process the data appropriately. A cond will be needed.

An exercise with mixed data

A playing card has a suit (one of hearts, diamonds, clubs, and spades) and a rank (an integer between 2 and 10, or one of Ace, Jack, Queen, King).

Jacks, Queens, and Kings are referred to as face cards.

In many card games, aces are worth 1, face cards worth 10, and other cards worth their rank. Write a function card-value that consumes the rank of a card, and produces its value.

Defining CardRank

The rank of a card is either a string (for Ace, Jack, Queen, King) or an integer between 2 and 10.

Even using the anyof notation that will be annoying to write multiple times. Let's define a new type instead. Here is its data definition:

```
;; A CardRank is (anyof Str Nat)
;; requires: if a Str, it is one of "Ace", "Jack", "Queen", "King"
;;           if a Nat, it is between 2 and 10, inclusive.
```

We can then write the contract:

```
;; card-value: CardRank → Nat
```

Completing card-value

Based on its data definition, any function that consumes a CardRank will likely need to include a cond expression to choose between the two types of possible values. Following the definition, we write the template:

```
;; cardrank-template: CardRank → Any
(define (cardrank-template r)
  (cond [(string? r) ... ]
        [ else ... ]))
```

```
;; card-value: CardRank → Nat
(define (card-value r)
  (cond [(string? r)
         (cond [(string=? r "Ace") 1]
               [else 10])]
        [else r]))
```

General equality testing

There are equality predicates for each type: e.g. =, string=?, boolean=?.

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.

Exercise: Rewrite card-value using equal?.

Additions to syntax

Syntax rule: an **expression** can be:

- a *value*,
- a single *constant*,
- a *function application*,
- a *conditional expression*, or
- a Boolean expression.

Recall: A **value** is a number, a string, or a Boolean value. A **constant** is a named **value**.

Design recipe for conditional functions

1. **Data analysis**. Figure out outcomes and inputs leading to each outcome. Influences examples, body, and tests. *(Not included in submission.)*
2. **Purpose**.
3. **Contract, including requirements.**
4. **Examples**. Check data for a few different situations.
5. **Definition**. Order the cases. Determine questions. Develop each question-answer pair.
6. **Tests**. Include at least one test for each answer.

Goals of this module

You should be comfortable with these terms: Boolean value, Boolean function, comparison, predicate.

You should be able to perform and combine comparisons to test complex conditions on numbers.

You should be able to trace programs using the substitution rules for and, or, and cond.

You should understand the syntax and use of a conditional expression.

You should use data analysis in the design recipe, and both black-box and white-box testing.

You should be able to write programs using strings and mixed data.

You should understand the (anyof . . .) notation and be able to use it in your own code.

You should understand the motivation of data-directed design.