Welcome to CS 115 (Fall 2019)

Web page (the main information source):

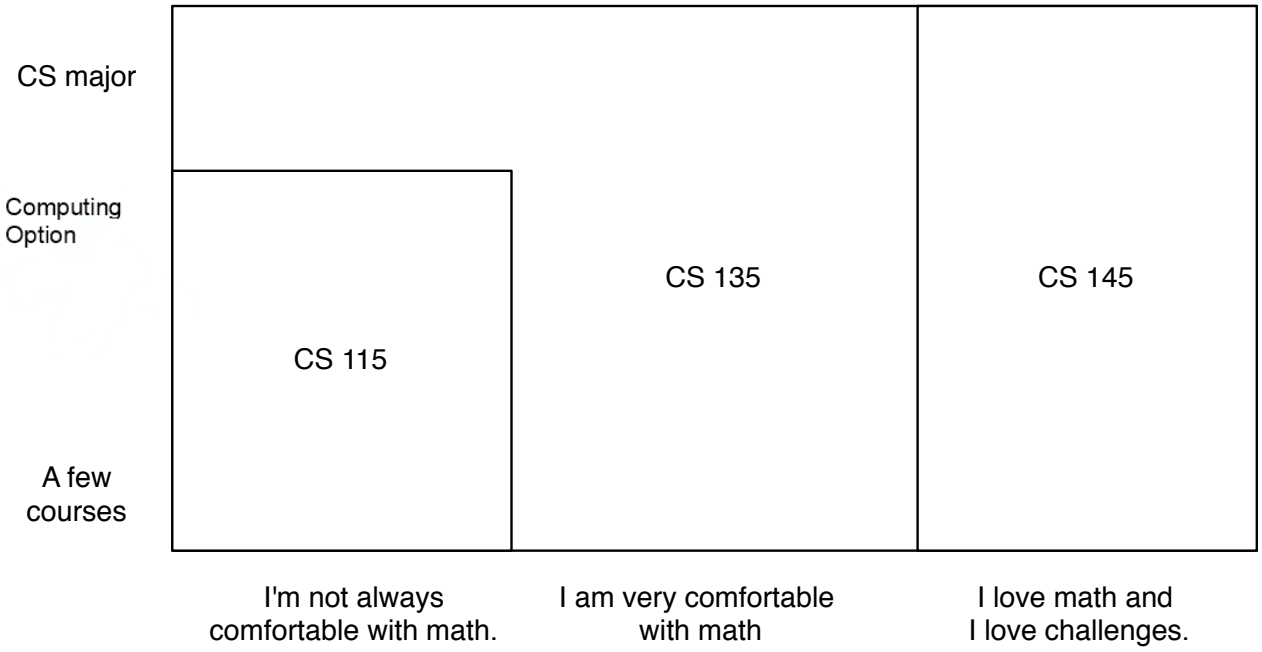
`http://www.student.cs.uwaterloo.ca/~cs115/`

Course Personnel: Contact information and office hours for all staff:

- instructors,
- ISAs (instructional support assistants),
- IAs (instructional apprentices), and
- the ISC (instructional support coordinator)

is available on the web page.

Am I in the right course?



Factors: interest in CS, comfort with math

Additional choice: CS 105 for non-math students

Course components

Lectures: Two 80 minutes lectures per week

Textbook: “How to Design Programs” (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi, First Edition (`http://www.htdp.org/2003-09-26`)

Presentation handouts: available on the web page and as a printed course pack from MC2018

Labs: 80 minutes using DrRacket v7.0 or higher (`http://www.DrRacket.org`)

Class Participation using Clickers

- There will be several multiple choice questions each lecture.
- You will earn 1 mark for any answer, plus 1 more if correct.
- Best 75% over term will be used to calculate grade.
- Questions **must** be answered in your own lecture.
- **Never** use another student's clicker.
- **You must register your clicker as part of A0.**
- *Purpose: To encourage active learning and provide real-time feedback (for the student and instructor).*

Assignments

- A0 - course logistics. **Must be completed for remaining assignments to be graded.**
- Assignments due weekly (roughly).
- You may use lab computers or your own computer.
- You must submit your assignments using MarkUs.
  - No email submissions will be accepted.
  - Submit your solutions early and often!
  - Basic tests are run on your submissions. Be sure to check the results.

Marking Scheme - check details on web page

- 20% Assignment
- 5% Participation
- 75% Exams - **You must pass the weighted exam average**
  - 30% Midterm
  - 45% Final

Under certain conditions, labs can earn an additional 3% bonus.

Grade Appeals: Review policy on course web page.

Academic Offenses: Review policy on course web page.

Work for your first week:

- Check your schedule for correct lecture and lab times.
- Attend lectures and lab.
- Read the Web pages for important course information (including staff, marking scheme, academic integrity policies, etc.).
- Read the Survival Guide (see the ”Resources” web page).
- Complete Assignment 0 (including registering your clicker).

Suggestions for success

- Keep up with the readings (keep ahead if possible).
- Attend lectures and take notes.
- Attend weekly labs and do the exercises.
- Start assignments early.
- Visit office hours to get help.
- Integrate exam study into your weekly routine.
- Keep on top of your workload.

More suggestions for success

- Follow our advice on approaches to writing programs (e.g. design recipe, templates).
- Go beyond the minimum required (e.g. do extra exercises).
- Maintain a “big picture” perspective: look beyond the immediate task or topic.
- Review your assignments and midterm: learn from your mistakes.
- Read email sent to your UW account.

Programming practice

Lectures prepare you for labs.

Labs prepare you for assignments.

Assignments prepare you for exams.

You must do your own work in this course; read the section on plagiarism in the CS 115 Survival Guide.

Role of programming languages

At the lowest level, a computer executes instructions in machine language. Machine language is designed for particular hardware, rather than being human-friendly and general.

People use high-level languages to express computation naturally and generally.

Programming language design

Imperative:

- frequent changes to data
- examples: machine language, Java, C++

Functional:

- computation of new values, not transformation of old ones
- examples: Excel formulas, LISP, ML, Haskell, Mathematica, XSLT, R (used in STAT 231)
- more closely connected to mathematics
- easier to design and reason about programs

Why start with Racket?

- used in education and research since 1975.
- minimal but powerful syntax
- tools easy to construct from a few simple primitives
- DrRacket, a helpful environment for learning to program
- nice transition to an imperative language, Python, in CS 116

Course goals

CS 115 **isn't** a course in programming in Racket.

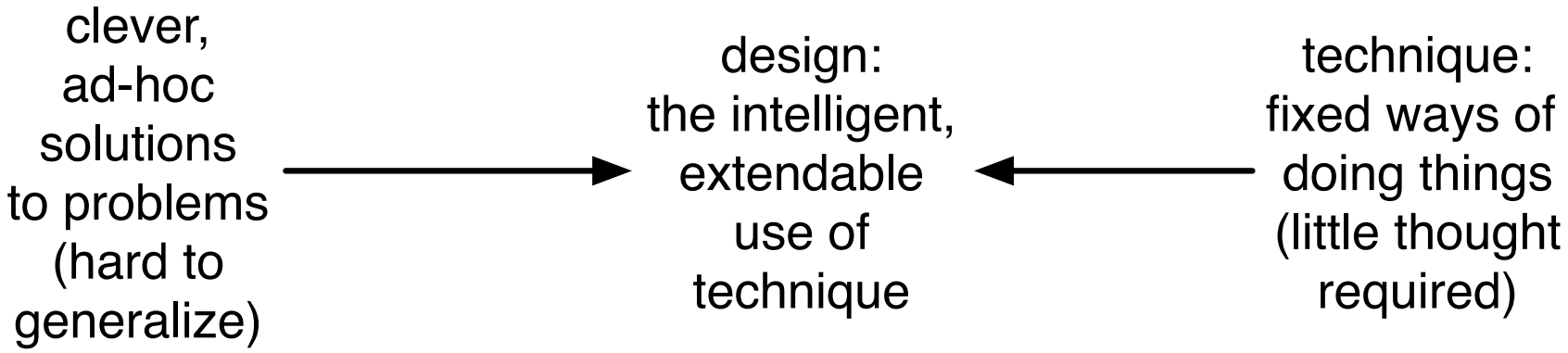
CS 115 **is** an introduction to computational thought (thinking precisely and abstractly).

CS 115 uses only a very small subset of Racket.

CS 116 will make the transition to an industry-oriented imperative language, Python.

Knowing two different programming paradigms will make it easier for you to extract the higher-level concepts being taught.

We will cover the whole process of designing programs.



Careful use of design processes can save time and reduce frustration, even with the fairly small programs written in this course.

Themes of the course

- design (the art of creation)
- abstraction (finding commonality, not worrying about details)
- refinement (revisiting and improving initial ideas)
- syntax (how to say it), expressiveness (how easy it is to say and understand), and semantics (the meaning of what's being said)
- communication (in general)

Goal: Learning how to think about solving problems using a computer.

Background terminology

Values are numbers or other mathematical objects.

For example, 5, $4/9$, 3.14159.

Expressions combine values with operators and functions.

For example, $5 + 2$, $\log_2(2 \cdot 8)$, $\frac{\sqrt{2}}{100 \cdot 5}$. Expressions can be simplified to values.

Functions generalize similar expressions.

Functions in mathematics

A function generalizes similar expressions.

$3^2 + 4 \cdot 3 + 2$

$6^2 + 4 \cdot 6 + 2$

$7^2 + 4 \cdot 7 + 2$

These are generalized by the function

$q(x) = x^2 + 4 \cdot x + 2.$

A mathematical **function definition** consists of

- the name of the function,
- its **parameters**, and
- a mathematical expression using the parameters.

Examples:

$f(x) = x^2$

$g(x, y) = x - y$

A **function application (or use)** supplies **arguments** for the parameters. Examples:

$g(3, 1)$

$g(f(2), g(3, 1))$

The mathematical expression is **evaluated** by substitution. The arguments are substituted into the expression in place of the parameters.

Example:

$g(3, 1) = 3 - 1 = 2$

We say that g **consumes** 3 and 1 and **produces** 2.

Mathematical expressions

For some mathematical functions, function applications are written in a different way.

In arithmetic expressions, the symbols for addition, subtraction, and division are put between numbers, for example, $3 - 2 + 4/5$.

Precedence rules (division before addition, left to right) specify the order of operation.

To associate a function with arguments in a way different from that given by the rules, parentheses are required: $(6 - 4)/(5 + 7)$.

Racket values, expressions, and functions

The simplest Racket values are numbers.

- Integers in Racket are unbounded.
- Rational numbers are represented exactly.
- Irrational numbers can be approximated by inexact Racket values.

We will learn about other Racket values later.

Racket expressions are built from values, operators, and functions.

Function applications in Racket

In a Racket function application, parentheses are put around the function name and the arguments.

To translate from mathematics to Racket

- move '(' to before the name of the function, and
- replace each comma with a space.

$g(3, 1)$ becomes (g 3 1)

$g(f(2), g(3, 1))$ becomes (g (f 2) (g 3 1))

There is one set of parentheses for each function application.

Converting an expression into Racket

As before, we put parentheses around the function and arguments.

$3 - 2$ becomes $(-\ 3\ 2)$

$3 - 2 + 4/5$ becomes $(+\ (-\ 3\ 2)\ (/\ 4\ 5))$

$(6 - 4) \cdot (3 + 2)$ becomes $(*\ (-\ 6\ 4)\ (+\ 3\ 2))$

In Racket, parentheses are used to associate arguments with functions, and are no longer needed for precedence.

Extra parentheses are harmless in mathematical expressions, but they are harmful in Racket. **Only use parentheses when necessary.**

The DrRacket environment

- designed for education, powerful enough for “real” use
- sequence of language levels keyed to textbook
- includes good development tools
- two windows: Interactions and Definitions
- Interactions window: a read-evaluate-print loop (REPL)

Built-in functions

Racket has many built-in functions, such as familiar functions from mathematics, other functions that consume numbers, and also functions that consume other types of data.

You can find the quotient of two integers using quotient and the remainder using remainder.

```
(abs -5)
(quotient 75 7)
(remainder 75 7)
```

Racket expressions causing errors

What is wrong with each of the following?

- `(* (5) 3)`
- `(+ (* 2 4)`
- `(5 * 14)`
- `(* + 3 5 2)`
- `(/ 25 0)`

Preventing errors

Syntax: the way we're allowed to express ideas

Syntax error when an expression cannot be interpreted using the syntax rules: `(+ - 3)`

Semantics: the meaning of what we say

Semantic error when an expression cannot be reduced to a value by application of substitution rules. It occurs when the expression is being simplified, after the syntax has been checked: `(/ 5 0)`

'Trombones fly hungrily' has correct syntax (spelling, grammar) but no meaning.

English syntax rules (e.g. a sentence has a subject, a verb, and an object) are not rigidly enforced.

Racket syntax rules are interpreted by a machine. They are rigidly enforced.

If you break a syntax rule, you will see an error message.

Syntax rule: a **function application** consists of the function name followed by the expressions to which the function applies, all in parentheses, or (functionname exp1 . . . expk).

We need rules for function names and expressions to complete this.

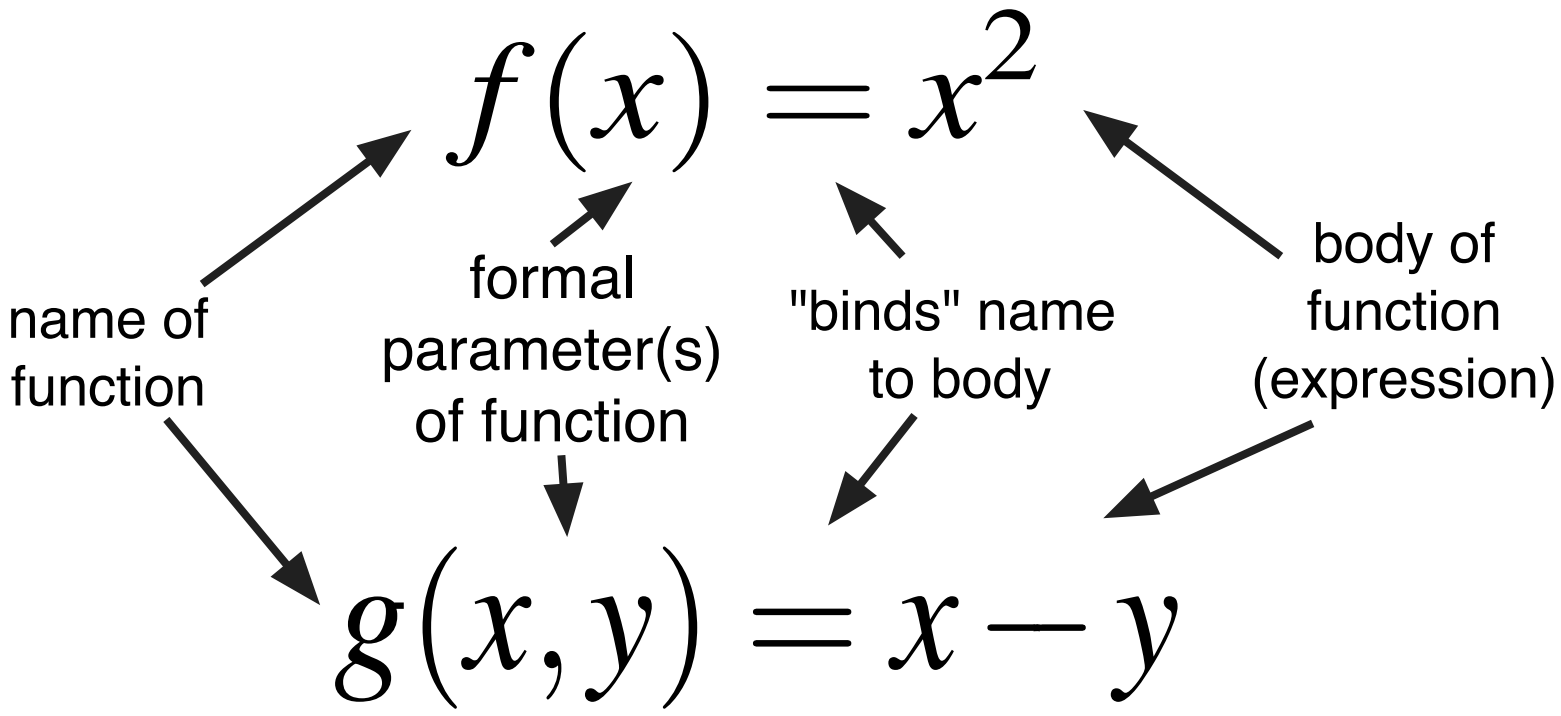
So far Racket is no better than a calculator or a spreadsheet.

A spreadsheet formula applies functions to data.

A spreadsheet application provides many built-in functions, but typically it is hard to construct your own.

Racket provides a modest number of built-in functions, but makes it easy to construct your own.

Defining functions in mathematics



Important observations:

- Changing names of parameters does not change what the function does: $f(x) = x^2$ and $f(z) = z^2$ have the same behaviour.
- The same parameter name can be used in different functions, as in f and g.
- The order of arguments must match the order of the parameters in the definition of the function: $g(3, 1) = 3 - 1$ but $g(1, 3) = 1 - 3$.
- Calling a function creates a new value.

Defining functions in Racket

Our definitions $f(x) = x^2$ and $g(x, y) = x - y$ become

```
(define (f x) (* x x))
(define (g x y) (- x y))
```

define is a **special form**; it looks like a Racket function, but not all of its arguments are evaluated.

It **binds** a name to an expression (which uses the parameters that follow the name).

A function definition consists of

- a name for the function,
- a list of parameters, and
- a single “body” expression.

The body expression typically uses the parameters together with other built-in and user-defined functions.

To give names to the function and parameters, we use identifiers.

Beginner Syntax rule: an **identifier** starts with a letter, and can include letters, numbers, hyphens, underscores, and a few other punctuation marks.

It cannot contain spaces or any of ( ) , { } [ ] ‘ ’ “ ”.

Syntax rule: **function definition** is of the form (define (id1 . . . idk) exp), where exp is an expression and each id is an identifier.

We can adjust the syntax rule for a function application to say that a function name is a built-in name or an identifier.

Important observations:

- Changing names of parameters does not change what the function does: (define (f x) (* x x)) and (define (f z) (* z z)) have the same behaviour.
- The same parameter name can be used in different functions, as in f and g.
- The order of arguments must match the order of the parameters in the definition of the function: (g 3 1) produces 2 but (g 1 3) produces -2.

DrRacket's Definitions window

- can accumulate definitions and expressions
- Run button loads contents into Interactions window
- can save and restore Definitions window
- provides a Stepper to evaluate expressions step-by-step
- features include: error highlighting, subexpression highlighting, syntax checking, bracket matching

Constants in Racket

The definitions $k = 3$ and $p = k^2$ become

```
(define k 3)
(define p (* k k))
```

The first definition binds the identifier k to the value 3.

In the second definition, the expression (* k k) is first evaluated to give 9, and then p is bound to that value.

Syntax rule: a **constant definition** is of the form (define id exp).

Defining constants allows us to give meaningful names to values (e.g. interest-rate, passing-grade, and starting-salary).

Usefulness of constants:

- Make programs easier to understand.
- Usually make future changes easier.
- May reduce typing (but may not).

The text uses the term variable to refer to constants, but their values cannot be changed.

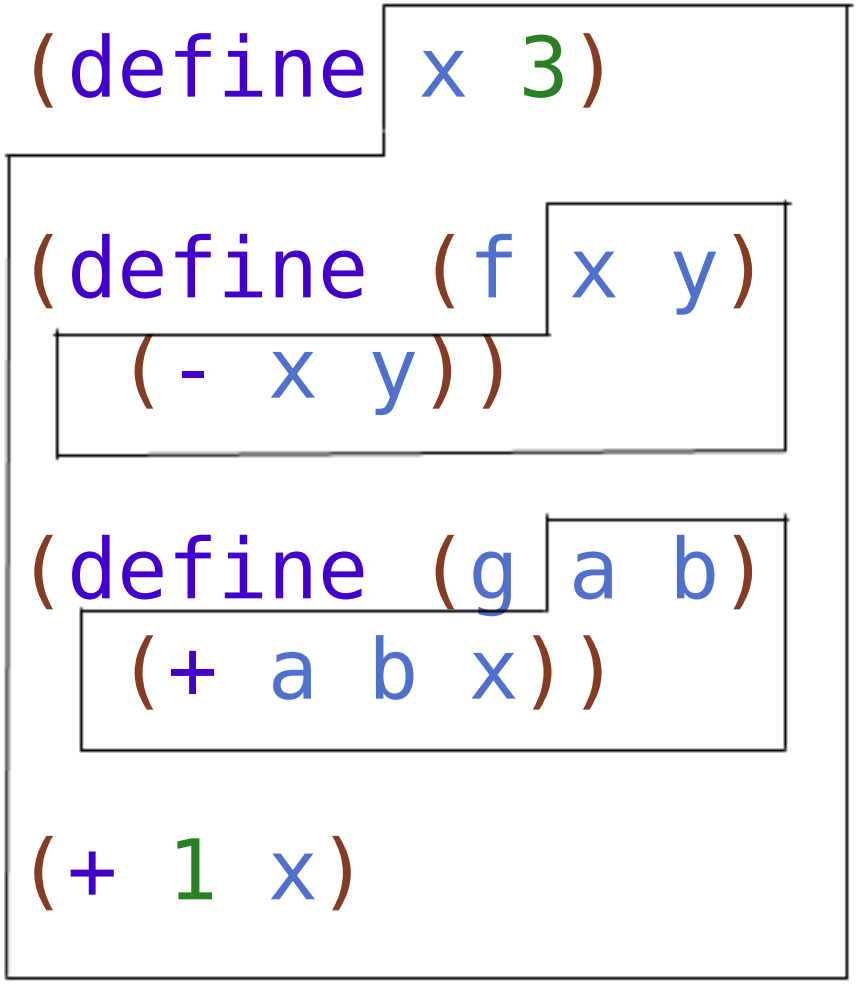
Programs in Racket

A **Racket program** is a sequence of definitions and expressions.

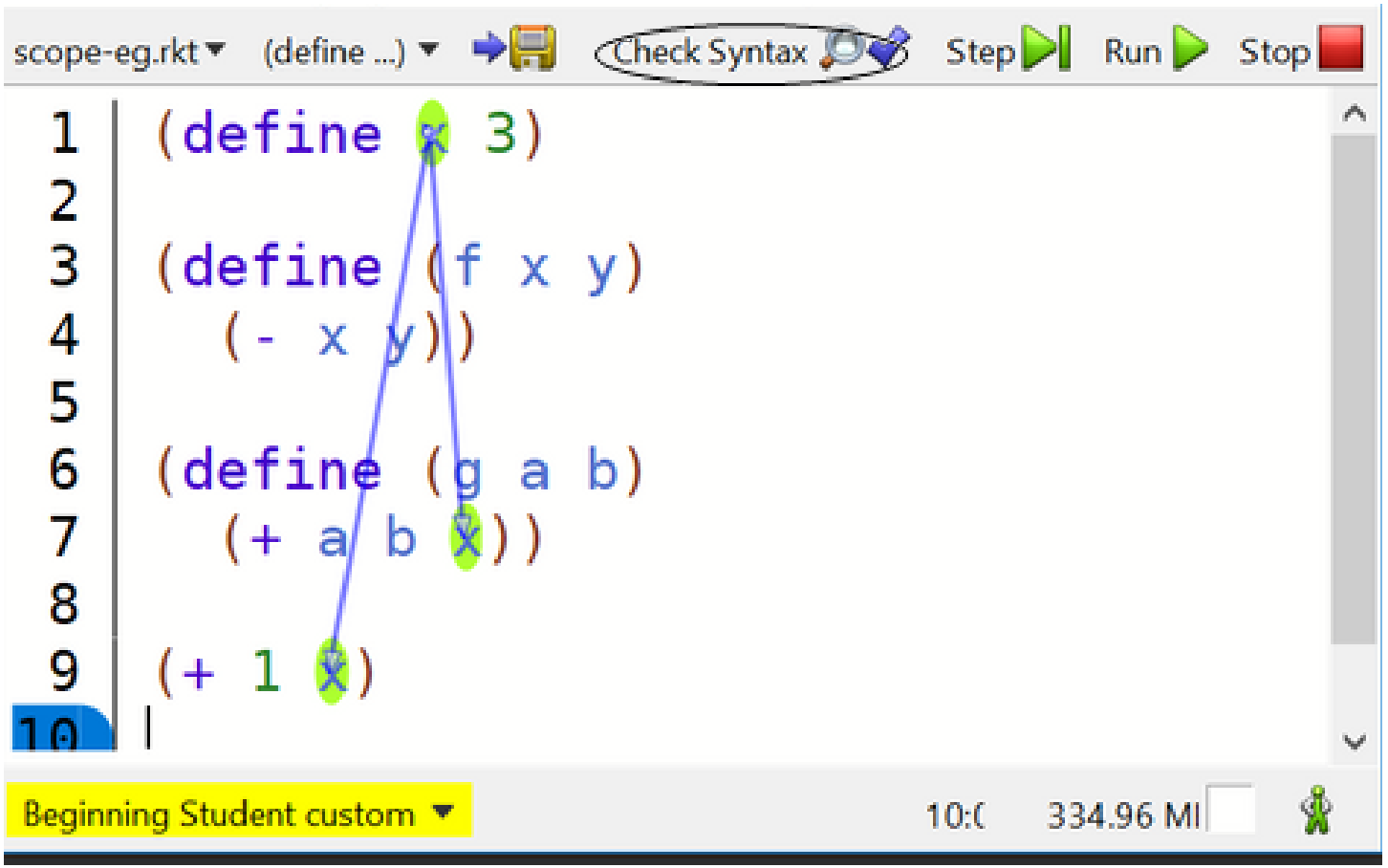
- The definitions are of functions and constants.
- The expressions typically involve both user-defined and built-in functions and constants.

Programs may also make use of **special forms** (which look like functions, but don't necessarily evaluate all their arguments).

Scope: where an identifier can be used



Scope in DrRacket



Identifiers and binding

Which of the following uses of x are allowed?

Can two functions use the same parameter name?

```
(define (f x) (* x x))
(define (g x y) (- x y))
```

Can a constant and a function parameter use the same name?

```
(define x 3)
(define (f x) (* x x))
```

Can two constants have the same name?

```
(define x 4)
(define x 5)
```

Can a constant and a function have the same name?

```
(define x 4)
(define (x y) (* 5 y))
```

Can the name of one function be the same as the parameter of another function?

```
(define (x y) (* 5 y))
(define (z x) (* 3 x))
```

Can a function name also be the name of one of its parameters?

```
(define (x x) (+ 1 x))                    ; Poor style
```

The importance of semantics

The English sentence “Cans fry cloud” is syntactically correct but has no semantic interpretation.

The English sentences “Students hate annoying professors”, “I saw her duck”, “You will be very fortunate to get this person to work for you”, and “Kids make nutritious snacks” are all ambiguous; each has more than one semantic interpretation.

Racket programs are unambiguous because of the rules of the language.

A semantic model

A semantic model of a programming language provides a method of determining the result of running any program.

A Racket program is a sequence of definitions and expressions.

Our model involves the simplification of the program using a series of steps (**substitution rules**).

For now, each step yields a valid Racket program.

A fully-simplified program is a sequence of definitions and values.

Values and Expressions

A Racket **value** is something which cannot be further simplified.

So far, the only things which are values are numbers. As we move through CS115, we will see many other types of values.

For example, 3 is a value, but (+ 3 5) and (f 3 2) are not.

A Racket **expression** is a value, a constant, or a function application.

Using substitution

We used substitution to evaluate a mathematical expression:

$$g(f(2), g(3,1)) = g(4, g(3,1)) = g(4,2) = 2$$

There are many possible ways to substitute, such as

$g(f(2), g(3,1)) = f(2) - g(3,1)$ and

$g(f(2), g(3,1)) = g(f(2), 2)$.

We will also use substitution for the Racket program:

```
(define (f x) (* x x))
(define (g x y) (- x y))
(g (f 2) (g 3 1))
```

Evaluating a Racket expression

Goal: a single sequence of substitution steps for any expression.

Constraints:

1. We first evaluate the arguments, and then apply the function to the resulting values.

2. When we have the choice among two or more substitutions, we take the **leftmost** (first) one.

These decisions do not change the final result.

We use the 'yields' symbol $\Rightarrow$ to show the result of a single step.

Substitution rules

Built-in function application: use mathematics rules.

$(+\ 3\ 5) \Rightarrow 8$

(quotient 75 7) $\Rightarrow$ 10

Constant: replace the identifier by the value to which it is bound.

(define x 3)

$(*\ \text{x}\ (+\ \text{x}\ 5))$

$\Rightarrow (*\ 3\ (+\ \text{x}\ 5))$

$\Rightarrow (*\ 3\ (+\ 3\ 5))$

$\Rightarrow (*\ 3\ 8) \Rightarrow 24$

Value: no substitution needed.

User-defined function application: for a function defined by (define (f x1 x2 . . . xn) exp), simplify a function application (f v1 v2 . . . vn) by replacing all occurrences of the parameter xi by the value vi in the expression exp.

Tricky points to remember:

- Each vi must be a **value**.
- Replace **all** occurrences in one step.

General rule: Anything that has been fully evaluated (e.g. a function definition) does not need to be repeated at the next step.

```
(define (f x) (* x x))
(define (g x y) (- x y))
(g (f 2) (g 3 1))
⇒ (g (* 2 2) (g 3 1))
⇒ (g 4 (g 3 1))
⇒ (g 4 (- 3 1))
⇒ (g 4 2)
⇒ (- 4 2)
⇒ 2
```

Tracing a program

Tracing: a step-by-step simplification by applying substitution rules.

Any expression that has been fully simplified does not need to appear in the next step of the trace.

Use tracing to check your work for semantic correctness.

If no rules can be applied but the program hasn't been simplified, there is an error in the program, such as (sqr 2 3).

Be prepared to demonstrate tracing on exams.

The Stepper may not use the same rules. Write your own trace to be sure you understand your code.

```
(define (term x y) (* x (sqr y)))
(term (- 5 3) (+ 1 2))
⇒ (term 2 (+ 1 2))
⇒ (term 2 3)
⇒ (* 2 (sqr 3))
⇒ (* 2 9)
⇒ 18
```

Goals of this module

You should be comfortable with these terms: function, parameter, application, argument, variable, expression, consume, produce, syntax, semantics, special form, bind, identifier, Racket program.

You should be able to define and use simple arithmetic functions, and to define constants.

You should know how to form expressions properly, and what DrRacket might do when given an expression causing an error.

You should understand the purposes and uses of the Definitions and Interactions windows in DrRacket.

You should be able to argue that your Racket code is syntactically correct, referring to Racket syntax rules.

You should understand the basic syntax of Racket, including rules for function application, function definition, and constant definition.

You should be able to trace the substitutions of a Racket program, using substitution rules for function applications and uses of constants.

Module 2: The design recipe

Readings:

- HtDP, Sections 1-3
- Survival Guide on assignment style and submission
- Style Guide

We are covering the same material as the text, but using different examples and style details. Lectures do not cover all the details, so keeping up with the readings is important.

Programs as communication

Every program is an act of communication:

- between you and the computer
- between you and yourself in the future
- between you and others

Comments

Comment: information important to humans but ignored by the computer

Code: anything that is not a comment; machine-readable

```
;; Constants for calendars
(define year-days 365) ; not a leap year
```

Use one semicolon to indicate that the rest of the line is a comment.

Use two semicolons to start a line that is a comment.

Do not use DrRacket's Comment Boxes.

Goals for software design

Misconception: correctness and speed only.

Partial list of goals for programs: compatible, composable, *correct*, durable, *efficient*, *extensible*, *flexible*, maintainable, portable, *readable*, *reliable*, reusable, *scalable*, usable, and useful.

The design recipe

- A development process that leaves behind a written explanation of the development.
- Use it for every function you write from now in CS115.
- Results in
  - reliability (the function has been tested)
  - readability (comments make it understandable)
- The basic recipe will be modified as we add new ingredients.

The design recipe components

The steps will appear in the following order:

- Purpose: Describes what the function calculates. The role of the parameters in the calculation should be made clear.
- Contract: Includes the type of arguments the function consumes and the value it produces. Includes additional required conditions on the inputs if needed.
- Examples: Shows the typical use of the function. Actual number of examples depends on the problem.

More design recipe components

- Definition: Racket code for the header and body of the function. Referred to as the ”implementation”.
- Tests: A set of inputs and expected outputs used to build evidence that the function meets its requirements. Actual number of tests depends on the problem and the implementation. Choose some tests that will help you identify problems in your code (special cases, boundaries, etc.).

Using the design recipe

Exercise: Write a function that sums the squares of two numbers.

Function header:

- Determine function name: sum-of-squares
- Determine parameters: p1 and p2, the two numbers

```
(define (sum-of-squares p1 p2) ...)
```

Purpose:

- Draft a sentence to explain what the function does.
- Include a basic function call
- Use the parameter names in your explanation to show how the produced value is related to the consumed parameter values.

```
;; (sum-of-squares p1 p2) produces the sum of
;;     the squares of p1 and p2
```

Contract:

- Determine what type each parameter should be.
- Determine what type of value is produced by the function.
- Follow the format precisely:

;; fun-name: type1 type2 ... typek $\rightarrow$ type-produced

Mathematically: sum-of-squares: $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, so we write:

;; sum-of-squares: Num Num $\rightarrow$ Num

Examples:

- Choose examples that show common usage of the function.
- Choose values for parameters and determine the answer.
- Develop the examples as: fn-call $\Rightarrow$ answer
- For example, (sum-of-squares 3 4) $\Rightarrow$ 25
- Another example, (sum-of-squares 0 -2.5) $\Rightarrow$ 6.25
- Convert to Racket code: (check-expect fn-call answer)

```
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 -2.5) 6.25)
```

Function definition:

- Write the body of the function.
- Combine with the header of the function to complete the implementation.

```
(define (sum-of-squares p1 p2)
  (+ (* p1 p1) (* p2 p2)))
```

Testing:

- **Tests:** a set of inputs and expected outputs, each designed to test a specific aspect of the function.
- Here, try different combinations for p1 and p2, for example, positive and negative, integer and real, zero.

```
(check-expect (sum-of-squares -1 -2) 5)
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -10 2.5) 106.25)
```

```
;; (sum-of-squares p1 p2) produces sum of squares of p1 and p2.
;; sum-of-squares: Num Num → Num
;; Examples:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)

(define (sum-of-squares p1 p2)
    (+ (* p1 p1) (* p2 p2)))

;; Tests for sum-of-squares:
(check-expect (sum-of-squares -1 -2) 5)
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -10 2.5) 106.25)
```

More design recipe details

We develop the design recipe in a different order than it appears:

Step 1: Develop **examples** and **tests.**

Step 2: Write the **Function header.**

Step 3: Draft the **purpose.**

Step 4: Determine the **contract.**

Step 5: Complete the **function body.**

Step 6: Finalize your **tests.**

Step 7: Run the program.

Step 8: Revise as needed.

Writing a purpose statement

- Explain what the function does.
- Try to use English and math notation rather than Racket code.
- Must include how the value of the parameters influences the result.
- Should not explain all the implementation details.
- Usually includes ”produces”.

Writing a contract

Our rules will be more strict than those used in the textbook.

So far, we have seen the following types (more will be added):

- Num: any numeric value
- Int: restricted to integers
- Nat: restricted to natural numbers (integers ≥ 0)
- Any: for any Racket value (no restrictions)

Additional contract requirements

If there are important constraints on the parameters that are not fully described by its type, add an additional **requires** section to ”extend” the contract.

```
;; (my-function a b c) ...
;; my-function: Num Num Num → Num
;; requires: 0 < a < b
;;                c must be nonzero
```

More about requirements

Sometimes, restrictions are needed to avoid errors:

```
;; (sqrt-shift x c) produces the squareroot of (x - c).
;; sqrt-shift: Num Num → Num
;; requires: x >= c
(define (sqrt-shift x c)
    (sqrt (- x c)))
```

Even more about requirements

Other times, restrictions are caused by the nature of the data:

```
;; (bump-grade g inc) produces g+inc, up to a maximum of 100
;; bump-grade: Num Num → Num
;; requires: 0 <= g <= 100
;;           0 <= inc <= 10
(define (bump-grade g inc)
    (min (+ g inc) 100))
```

Writing examples

- Choose “typical” inputs for examples to show what the function does.
- Determine, by hand, what the answer should be.
- More than one example may be needed.
- Examples should be chosen **before** you write your implementation.
- Examples do not need to cover all possibilities. We have tests for that.

Writing tests

- Choose a situation to test (target a specific aspect of the problem or of the code)
- Choose specific values for the function parameters
- Determine (by hand) the expected result (the known answer)
- Compare the expected to the actual value produced by your function.
- Tests don't need to be "big". Smaller is often better, because it is easier to figure out what went wrong if they fail.

Testing builds on the examples - tests are more complete.

Warnings about testing your functions

- Never use the function to determine the expected value!
- Never cut-and-paste from interactions window to definitions window.
- Some students do not include enough tests.
- Some students have lots of tests - but they mostly cover the same situation.
- Determining the correct number of tests to use is a judgement call. You have to determine it based on the problem and the implementation.

Completing examples and tests in DrRacket

The student languages offer a convenient testing method:

```
(check-expect fun-application value-expected)
```

For example:

```
(check-expect (sum-of-squares 3 4) 25)

(check-expect (abs -5) 5)
```

Note: check-expect only works with exact numbers.

The following code will result in an error.

```
(define (circle-area r) (* pi r r))
(check-expect (circle-area 1) pi)
```

When using expected or actual values which are inexact or approximations, use check-within.

(check-within fun-application value-expected tol)

checks that abs(fun-application − value-expected) $\leq$ tol

A tolerance of .00001 should typically suffice (unless told otherwise on an assignment).

(check-within (circle-area 1) pi .00001)

(check-within (sqrt 2) 1.414 0.001)

Design recipe style guide

Note that in these slides, sections of the design recipe are often omitted or condensed because of space considerations.

Consult the course style guide on the course webpage before completing your assignments.

In particular, the style guide provides more guidance on testing different types of values.

The string data type (Str)

A **Str** is a value made up of letters, numbers, blanks, and punctuation marks, all enclosed in quotation marks.

Examples: "hat", "This is a string.", and "Module 2".

String functions:

(string-append "now" "here") ⇒ "nowhere"

(string-length "length") ⇒ 6

(substring "caterpillar" 5 9) ⇒ "pill"

(substring "cat" 0 0) ⇒ ""

(substring "nowhere" 3) ⇒ "here"

More about strings

- Strings start from position 0.
- (substring t start stop) produces a string of length stop − start, assuming that 0 <= start <= stop <= (string-length t).
- Special value: the empty string "" has length 0.
- Functions can consume and produce string values.
- See the course Web site for more Str documentation (not in the textbook).

Exchanging front and back of a string

swap-parts will exchange the front and back of a string.

The *front* is the first half of the string.

The *back* is the second half of the string.

If the length of the string is odd, we'll make the front shorter by including the middle character with the back.

```
;; (swap-parts s) produces a new string like s, with front and back
;;    parts reversed.
;; swap-parts: Str → Str
;; Examples:
(check-expect (swap-parts "angle") "glean")
(check-expect (swap-parts "potshots") "hotspots")

(define (swap-parts s) ...)
```

Subtasks: finding the midpoint, extracting the front, extracting the back.

Helper functions

A **helper function** is a function that is used in the primary function to

- generalize similar expressions,
- express repeated computations,
- perform smaller tasks required by your solution, or
- improve readability of the code

Choose meaningful function names for all functions and parameters, including helper functions (do not call it helper!).

Put helper functions before the primary function.

You do not need to include tests for helper functions on assignments.

Finding the midpoint

```
;; (mid t) produces the integer part of (string-length t)/2.
;; mid: Str → Nat
;; Examples:
(check-expect (mid "cs115") 2)
(check-expect (mid "Hi") 1)

(define (mid t) (quotient (string-length t) 2))

;; Tests for mid
(check-expect (mid "") 0)
(check-expect (mid "A") 0)
```

Extracting the front

```
;; (front-part s) produces the front part of s.
;; front-part: Str → Str
;; Examples:
(check-expect (front-part "angle") "an")
(check-expect (front-part "potshots") "pots")

(define (front-part s) (substring s 0 (mid s)))

;; Tests for front-part
(check-expect (front-part "") "")
(check-expect (front-part "Z") "")
```

Extracting the back

```
;; (back-part s) produces the back part of s.
;; back-part: Str → Str
;; Examples:
(check-expect (back-part "angle") "gle")
(check-expect (back-part "potshots") "hots")

(define (back-part s) (substring s (mid s)))

;; Tests for back-part
(check-expect (back-part "") "")
(check-expect (back-part "B") "B")
```

```
;; (swap-parts s) produces a new string like s, with front and back
;;    parts reversed.
;; swap-parts: Str → Str
;; Examples:
(check-expect (swap-parts "angle") "glean")
(check-expect (swap-parts "potshots") "hotspots")

(define (swap-parts s)
   (string-append (back-part s) (front-part s)))

;; Tests for swap-parts
(check-expect (swap-parts "") "")
(check-expect (swap-parts "Z") "Z")
```

Warnings about contracts

Our contracts are comments. They are not verified by the Racket interpreter when a function is called.

For example, if we call (swap-parts 10), we get an error, but it may not be where you expect.

```
(swap-parts 10)
⇒ (string-append (back-part 10) (front-part 10))
⇒ (string-append (substring 10 (mid 10)) (front-part 10))
⇒ (string-append (substring 10 (quotient (string-length 10)))
                 (front-part 10))
⇒ ERROR
```

More about contracts

The error occurs when we cannot simplify our expression further, not simply because the contract was violated.

In CS 115, you should assume data provided in a function call wlll satisfy the contract unless you are told otherwise.

Dynamic typing vs Static typing

Some programming languages (like Java and C) require that the type of each parameter is specified when the function is defined. This is called *static typing*.

Racket does not require us to specify types ahead of time. This is called *dynamic typing*. Functions can be called with values of any type. Errors only arise when we try to use the values in ways they cannot be used (for example, when we try to take the length of an integer).

Computing a phone bill

cell-bill consumes the numbers of daytime and evening minutes used and produces the total charge for minutes used.

Details of the phone plan:

- 100 free daytime minutes
- 200 free evening minutes
- 1 dollar per minute charge for each additional daytime minute
- 50 cent per minute charge for each additional evening minute

How can we remember the meaning of each number?

Using constants

```
;; constants for phone plan
;; Free limits
(define day-free 100)
(define eve-free 200)
;; Rates per minute
(define day-rate 1)
(define eve-rate .5)
```

Use constants for readability and flexibility.

Put them before helper functions and primary functions.

```
;; (cell-bill day eve) produces cell phone bill for day daytime
;;    minutes and eve evening minutes used.
;; cell-bill: Nat Nat → Num
;; Examples:
(check-expect (cell-bill 101 0) 1)
(check-expect (cell-bill 99 0) 0)
(check-expect (cell-bill 0 199) 0)
(check-expect (cell-bill 0 202) 1)
(check-expect (cell-bill 150 300) 100)

(define (cell-bill day eve) ...)
```

```
;; (charges-for minutes freelimit rate) produces charges for minutes,
;; given the rate per minute past the freelimit.
;; charges-for: Nat Nat Num → Num
;; requires: rate >= 0
;; Examples:
(check-expect (charges-for 101 100 5) 5)
(check-expect (charges-for 99 100 34) 0)

(define (charges-for minutes freelimit rate)
  (max 0 (* (- minutes freelimit) rate)))

;; Tests for charges-for:
(check-expect (charges-for 100 100 5) 0)
```

Completing cell-bill

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

;; Tests for cell-bill
(check-expect (cell-bill 100 0) 0)
(check-expect (cell-bill 0 200) 0)
(check-expect (cell-bill 50 175) 0)
(check-expect (cell-bill 100 200) 0)
```

Goals of this module

You should be comfortable with these terms: comment, code, contract, requirements, purpose, examples, definition, function header, body, tests, helper function.

You should know the types that are allowed in contracts so far.

You should understand the reasons for each of the components of the design recipe, the order in which they appear, and the order in which they should be created.

You should know when to use check-expect and when to use check-within.

You should start to use the design recipe and appropriate coding style for all Racket programs you write.

You should look for opportunities to use helper functions and constants to structure your programs, and gradually learn when and where they are appropriate.

You should be able to use strings in labs, assignments, and exams.

Module 3: New types of data

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 $\rightarrow$ 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]. . .)

$\Rightarrow$ (cond [exp1 exp2]. . .).

(cond [true exp]. . .) $\Rightarrow$ exp.

(cond [else exp]) $\Rightarrow$ 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 (check-divide n)
  (cond [(zero? n) "Undefined"] [else (/ 1 n)]))
(check-divide 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 bonus0 0)
(define bonus1 100)
```

```
;; bonus: Num → Num
(define (bonus diff)
    (cond [(and (<= bonus0 diff) (< diff bonus1)) diff]
          [(<= bonus1 diff) (* 2 diff)]
          [(< diff 0) 0]))
```

Nicer ordering:

```
(define (bonus diff)
   (cond [(< diff bonus0) 0]
         [(< diff bonus1) 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 0) 0]
    [(< diff 100) 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.

```
;; check-divide: Num → (anyof "Undefined" Num)
(define (check-divide 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 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)

Generalized odd

gen-odd? consumes a number (which may or may not be an integer) or a string. It produces true if the input is an odd integer or a string whose length is an odd integer, and false otherwise. In particular, it produces false for a number which is not an integer.

What is the contract?

How many examples and tests are needed to cover all cases?

A function that consumes mixed data contains a cond with one question for each type of data.

Racket provides predicates to identify data types, such as integer?, number? and string?.

```
;; gen-odd?: (anyof Num Str) → Bool
(define (gen-odd? info)
  (cond
    [(integer? info) ...]
    [(number? info) ...]
    [else ...]))
```

Completing generalized odd

```
(define (gen-odd? info)
  (cond
    [(integer? info)
     (cond [(odd? info) true] [else false])]
    [(number? info)
     false]
    [else
     (cond [(odd? (string-length info)) true]
           [else false])]))
```

Simplifying generalized odd

```
(define (gen-odd? info)
  (cond
    [(integer? info) (odd? info)]
    [(number? info) false]
    [else (odd? (string-length info))]))
```

Generalized odd without cond

```
(define (gen-odd? info)
  (or (and (integer? info) (odd? info))
      (and (string? info) (odd? (string-length info)))))
```

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.

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.

Module 4: Lists

Readings: HtDP, Sections 9, 10.

So far, we have written and used functions that consume a small amount of data: a few numbers, strings, or Boolean values. We have known exactly how many pieces of information we had.

Often, though, we don't know exactly how much data we need. We may have a collection of ids of students interested in adding a full course, or possible names for a new pet, but not know exactly how many of those values we have.

Lists are the main tool used in Racket to work with unbounded data.

We can store student information in a list (even if we do not know the total number of students ahead of time) and we can use built-in list functions to retrieve information from that list.

We will write functions that can consume and produce lists.

Before we can do that, however, we need to define what we mean by a list.

A data definition for lists

What is a good data definition for a list of numbers?

A list has a first item, a second item, a third item, etc, and ends with a last item. Unfortunately, this definition is pretty vague.

We need a definition that refers to a list of any length:

If we take away one number from a nonempty list of numbers, what remains is a smaller list of numbers.

We will use this idea for our definition.

A list is a recursive structure - it is defined in terms of a smaller list.

- A list of 3 numbers is a number followed by a list of 2 numbers.
- A list of 2 numbers is a number followed by a list of 1 number.
- A list of 1 number is a number followed by a list of 0 numbers.

A list of zero numbers is special. We call it the empty list.

Informally, a list of numbers is either empty or it consists of a first number followed by the rest of the list (which is a smaller list of numbers).

Recursive definitions

A recursive definition includes at least one reference to itself. This is called a recursive case. *The rest of a list is a list.*

A recursive definition includes at least one case which does not reference itself. This is called a base case. *A list may be empty.*

Though we didn't state it explicitly, we have seen this idea before.

A *binary mathematical expression* is either

- a number, or
- two *binary mathematical expressions* combined with +,-,*,/.

A recursive data definition of a list

;; A **(listof Any)** is one of:

;; * empty

;; * (cons Any (listof Int))

Informally: a list is either empty, or it consists of a **first** value followed by a list (the **rest** of the list).

This is a **recursive** definition, with one **base** case, and one **recursive** (self-referential) case.

Lists are the main data structure in standard Racket.

The empty constant

- Racket also includes another constant '(), which is equivalent to empty.
- You may use either representation of the empty list. We will use empty in the course notes.
- You may use the "Show Details" option when choosing your language level to set your preferred representation for the empty list (and for the boolean constants, as previously discussed).

Constructing lists

Any nonempty list is constructed from an item and a smaller list using cons.

In constructing a list step by step, the first step will always be consing an item onto an empty list.

```
(cons "blue" empty)
(cons "red" (cons "blue" empty))
(cons (sqr 2) empty)
(cons (cons 3 (cons true empty)) (cons (make-posn 1 3) empty))
```

Deconstructing lists

(first (cons "a" (cons "b" (cons "c" empty))))

$\Rightarrow$ "a"

(rest (cons "a" (cons "b" (cons "c" empty))))

$\Rightarrow$ (cons "b" (cons "c" empty))

Substitution rules:

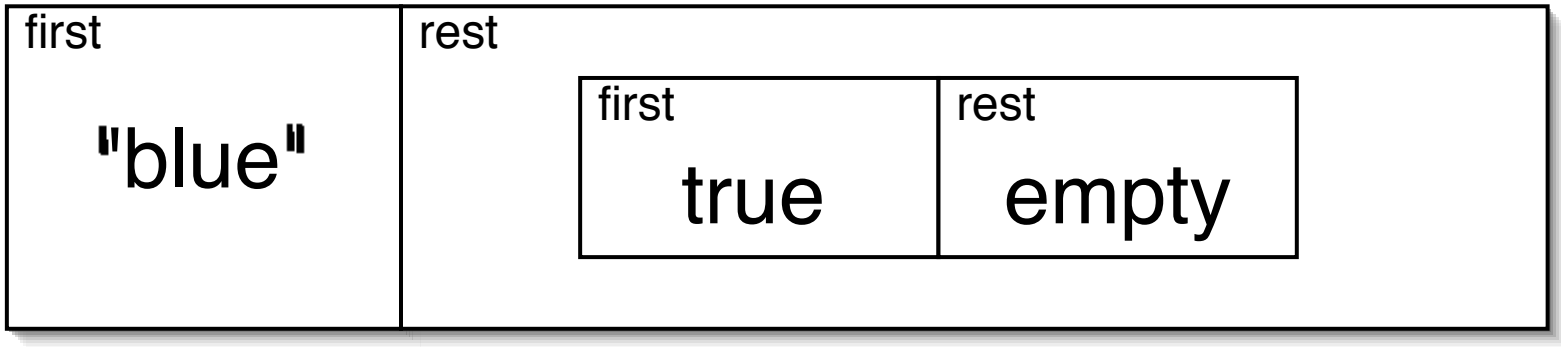
(first (cons elt alist)) $\Rightarrow$ elt

(rest (cons elt alist)) $\Rightarrow$ alist

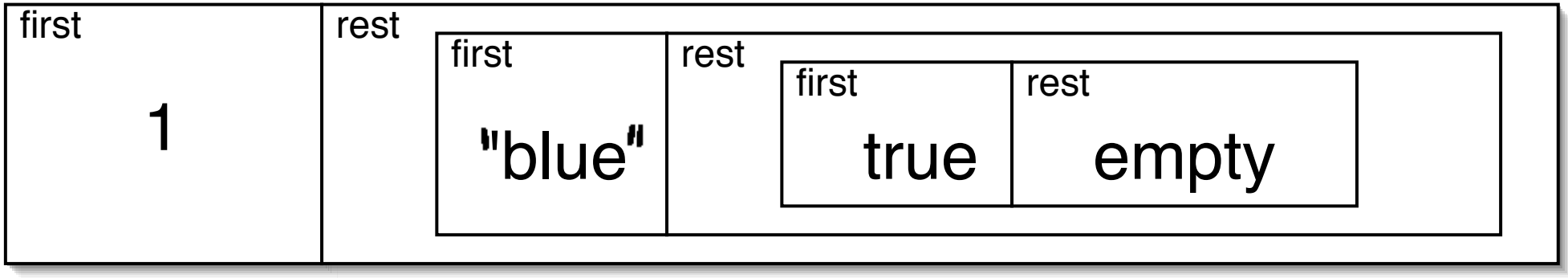
The functions consume nonempty lists only.

Nested boxes visualization

(cons "blue" (cons true empty))

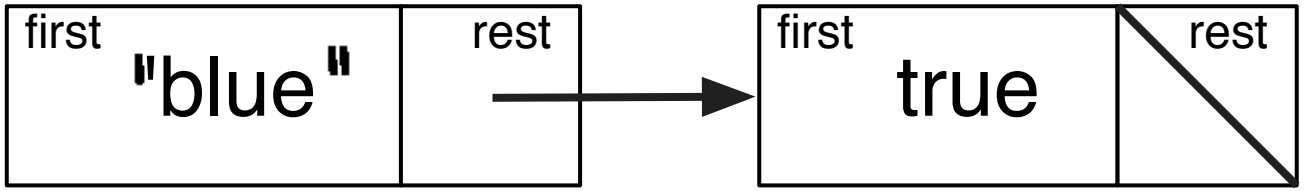


(cons 1 (cons "blue" (cons true empty)))

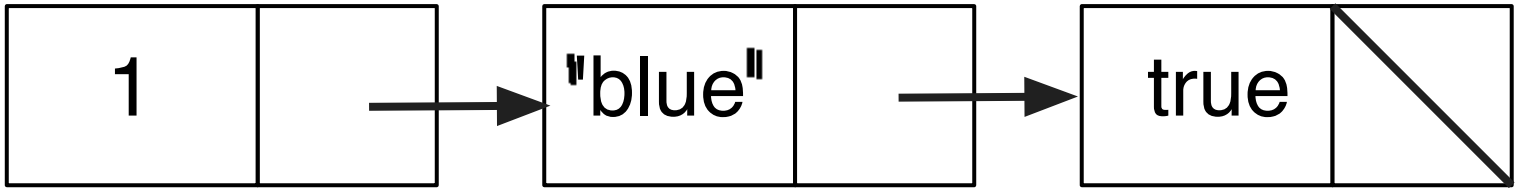


Box-and-pointer visualization

(cons "blue" (cons true empty))



(cons 1 (cons "blue" (cons true empty)))



Extracting values

```
(define mylist (cons 1 (cons "blue" (cons true empty))))
```

What expression evaluates to:

- 1 ?
- "blue" ?
- (cons true empty) ?
- true ?
- empty ?

Recursive data definition of a list of integers

;; A **(listof Int)** is one of:

;; * empty

;; * (cons Int (listof Int))

Informally: a list of integers is either empty, or it consists of a **first** integer followed by a list of integers (the **rest** of the list).

This is a **recursive** definition, with a **base** case, and a **recursive** (self-referential) case.

Which of these are lists of integers?

empty

(cons 5 empty)

(cons −1 (cons 5 empty))

(cons −1 (cons 5 3))

Templates and data-directed design

One of the ideas of the HtDP textbook is that the definition (or form) of data may mirror the form of a function that consumes that data.

A template is a general pattern for a function that consumes a specific type of data. It will be a starting point for our function implementations.

The list template will be developed from the recursive definition of a list. From the definition of a list, it can be empty or not. A function that consumes a list needs to determine if the list is empty before it can do anything else.

List template

We can use the built-in type predicates empty? and cons? to distinguish between the two cases in the data definition.

```
;; loi-template: (listof Int) → Any
(define (loi-template aloi)
  (cond
    [(empty? aloi) ...]
    [(cons? aloi) ...]))
```

The second test can be replaced by else.

We can also use the list selectors to expand the template.

Developing the template

```
;; loi-template: (listof Int) → Any
(define (loi-template aloi)
  (cond
    [(empty? aloi) ...]
    [else (... (first aloi) ... (rest aloi) ...)]))
```

Next idea: since rest aloi is also a list of integers, we can apply loi-template to it. This recursive application of a function solves the problem for the rest of the list. It needs to be combined with the first value in the list to solve the original problem.

Here is the resulting template for a function that consumes a list, which matches the data definition:

```
;; loi-template: (listof Int) → Any
(define (loi-template aloi)
  (cond
    [(empty? aloi) ...]
    [else (... (first aloi) ...
           ... (loi-template (rest aloi)) ...)]))
```

A function is **recursive** when the body involves an application of the same function (it uses **recursion**).

Example: my-length

```
;; (my-length aloi) produces the number of integers in aloi.
;; my-length: (listof Int) → Nat
;; Examples:
(check-expect (my-length empty) 0)
(check-expect (my-length (cons 3  (cons -2 empty))) 2)

(define (my-length aloi)
  (cond
    [(empty? aloi) 0]
    [else (+ 1 (my-length (rest aloi)))]))
```

Tracing my-length

```
(my-length (cons 3 (cons -2 empty)))
⇒ (cond [(empty? (cons 3 (cons -2 empty))) 0]
        [else (+ 1 (my-length (rest (cons 3 (cons -2 empty)))))])
⇒ (cond [false 0]
        [else (+ 1 (my-length (rest (cons 3 (cons -2 empty)))))])
⇒ (cond [else (+ 1 (my-length
                    (rest (cons 3 (cons -2 empty))))))])
⇒ (+ 1 (my-length (rest (cons 3 (cons -2 empty)))))
⇒ (+ 1 (my-length (cons -2 empty)))
```

$\Rightarrow$ (+ 1 (cond [(empty? (cons −2 empty)) 0]
[else (+ 1 . . .)]))

$\Rightarrow$ (+ 1 (cond [false 0] [else (+ 1 . . .)]))

$\Rightarrow$ (+ 1 (cond [else (+ 1 . . .)]))

$\Rightarrow$ (+ 1 (+ 1 (my-length (rest (cons −2 empty)))))

$\Rightarrow$ (+ 1 (+ 1 (my-length empty)))

$\Rightarrow$ (+ 1 (+ 1 (cond [(empty? empty) 0] [else (+ 1 . . .)])))

$\Rightarrow$ (+ 1 (+ 1 (cond [true 0] [else (+ 1 . . .)])))

$\Rightarrow$ (+ 1 (+ 1 0))

$\Rightarrow$ (+ 1 1)

$\Rightarrow$ 2

The trace condensed

(my-length (cons 3 (cons −2 empty)))

⇒ (+ 1 (my-length (cons −2 empty)))

⇒ (+ 1 (+ 1 (my-length empty)))

⇒ (+ 1 (+ 1 0))

⇒ 2

This condensed trace shows how the application of a recursive function leads to an application of the same function to a smaller list, until the **base case** is reached.

Condensed traces

The full trace contains too much detail, so we define the condensed trace with respect to a recursive function my-fn to be the following lines from the full trace:

- Each application of my-fn, showing its arguments;
- The result once the base case has been reached;
- The final value (if above expression was not simplified).

From now on, for the sake of readability, we will tend to use condensed traces, and even ones where we do not fully expand constants.

If you wish to see a full trace, you can use the Stepper to generate one.

But as we start working on larger and more complex forms of data, it becomes harder to use the Stepper, because intermediate expressions are so large.

Structural recursion

In the template, the form of the code matches the form of the data definition of what is consumed.

The result is **structural recursion**.

There are other types of recursion which we will cover in CS116.

You are expected to write structurally recursive code in CS115.

Using the templates will ensure that you do so.

Design recipe refinements

Only changes are listed here; the other steps stay the same.

Do this once per self-referential data type:

Data analysis and design: This part of the design recipe may contain a self-referential data definition, either a new one or one we have seen before.

At least one clause (possibly more) in the definition must not refer back to the definition itself; these are base cases.

You only need to include new definitions in this step. You do not need to submit the basic list definition.

Template: The template follows directly from the data definition.

The overall shape of the template will be a cond expression with one clause for each clause in the data definition.

Self-referential data definition clauses lead to recursive expressions in the template.

Base case clauses will not lead to recursion.

You only need to submit templates with your solutions when explcitly asked.

The per-function part of the design stays as before.

The (listof Int) template revisited

```
;; A (listof Int) is one of:
;; * empty
;; * (cons Int (listof Int))

;; loi-template: (listof Int) → Any
(define (loi-template aloi)
  (cond
    [(empty? aloi) ...]
    [else (... (first aloi) ... (loi-template (rest aloi)) ...)]))
```

There is nothing in the data definition or template that depends on integers. We could substitute Str throughout and it still works.

```
;; A (listof Str) is one of:
;; * empty
;; * (cons Str (listof Str))

;; los-template: (listof Str) → Any
(define (los-template alos)
  (cond
    [(empty? alos) ...]
    [else (... (first alos) ... (los-template (rest alos)) ...)]))
```

Types used in contracts

We have now seen lots of different types of data used in our functions. All can be used in our contracts:

- Types which are built into Racket. For example, Num, Int, Str, and Bool.
- Types defined only inside a data definition, such as (listof Int)

listof notation in contracts

As we noted, the data definitions for (listof Int) and (listof Str) are the same, except one uses Int and one uses Str.

We'll use (listof τ) in contracts, where τ may be replaced with any type. Examples: (listof Num), (listof Bool), (listof (anyof Num Str)), and (listof Any).

Always use the singular form for the type, e.g. (listof Num) not (listof Nums).

Always replace τ with the most specific type available.

Templates

When we use (listof τ) (replacing τ with a specific type, of course), we will assume the following generic template. You do **not** need to write a new template each time.

```
;; listof-τ-template: (listof τ) → Any
(define (listof-τ-template lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst) ... (listof-τ-template (rest lst)) ...)]))
```

You will use this template many times!

Templates as generalizations

Templates reduce the need to use examples as a basis for writing new code (though we will still need examples).

You can think of a template as providing the basic shape of the code as suggested by the data definition.

As we learn and develop new data definitions, we will develop new templates.

Use the templates!

Design recipe modifications, continued

Examples/Tests: Exercise all parts of the data definition; for lists, at least one base and one recursive case, though more may be needed.

Body: Use examples to fill in the template.

Base case(s): First fill in the cond-answers for the cases which don't involve recursion.

Recursive case(s): For each example, determine the values provided by the template (the first item and the result of applying the function to the rest). Then figure out how to combine these values to obtain the value produced by the function.

Example: count-apples

```
;; (count-apples alos) produces the number of occurrences of
;;    "apple" in alos.
;; count-apples: (listof Str) → Nat
;; Examples:
(check-expect (count-apples empty) 0)
(check-expect (count-apples (cons "apple" empty)) 1)
(check-expect (count-apples (cons "pear" (cons "peach" empty))) 0)

(define (count-apples alos) ...)
```

Using the template

The template is a starting point:

- You may need more than one base,
- You may need more than one recursive case, as actions may depend on the value of the first item in the list.

The specifics depend on the problem itself.

Generalizing count-apples

We can make this function more general by providing the string to be counted.

```
;; (count-given alos target) produces the number of occurrences
;;    of target in alos.
;; count-given: (listof Str) Str → Nat
;; Examples:
(check-expect (count-given empty "pear") 0)
(check-expect (count-given (cons "apple" empty) "apple") 1)
(check-expect (count-given
                    (cons "pear" (cons "peach" empty)) "pear") 1)
```

Extra information in a list function

By modifying the template to include one or more parameters that "go along for the ride" (that is, they don't change), we have a variant on the list template.

```
;; extra-info-list-template: (listof Any) Any → Any
(define (extra-info-list-template alist info)
  (cond
    [(empty? alist) ...]
    [else (... (first alist)... info ...
               (extra-info-list-template (rest alist) info) ...)]))
```

Built-in list functions

A closer look at my-length reveals that it will work just fine on lists of type (listof Any). It is the built-in Racket function length.

The built-in pedicate member? consumes an element of any type and a list, and returns true if and only if the element appears in the list.

You may now use cons, first, rest, empty?, cons?, length, and member? in the code that you write.

Do not use other built-in functions until we have learned about them.

Producing lists from lists

negate-list consumes a list of numbers and produces the same list with each number negated (3 becomes -3).

For example:

(negate-list empty) $\Rightarrow$ empty

(negate-list (cons 2 (cons -12 empty)))

$\Rightarrow$ (cons -2 (cons 12 empty))

Building negate-list from template

```
;; (negate-list alon) produces a list formed by negating
;;    each item in alon.
;; negate-list: (listof Num) → (listof Num)
;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
              (cons -2 (cons 12 empty)))
(define (negate-list alon)
  (cond [(empty? alon) ...]
        [else (... (first alon) ... (negate-list (rest alon)) ...)]))
```

negate-list completed

```
(define (negate-list alon)
  (cond
    [(empty? alon) empty]
    [else (cons (- (first alon)) (negate-list (rest alon)))]))
```

Condensed trace of negate-list

(negate-list (cons 2 (cons −12 empty)))

⇒ (cons (− 2) (negate-list (cons −12 empty)))

⇒ (cons −2 (negate-list (cons −12 empty)))

⇒ (cons −2 (cons (− −12) (negate-list empty)))

⇒ (cons −2 (cons 12 (negate-list empty)))

⇒ (cons −2 (cons 12 empty))

Removing elements from a list

```
;; (my-remove-all n alon) produces list with all occurrences of n
;;    removed from alon.
;; my-remove-all: Num (listof Num) → (listof Num)
;; Examples:
(check-expect (my-remove-all 2 empty) empty)
(check-expect (my-remove-all  2 (cons 2 (cons 12 empty)))
                        (cons 12 empty))
(define (my-remove-all n alon)
   (cond
      [(empty? alon) ...]
      [else (... (first alon) ... (my-remove-all n (rest alon))...)]))
```

Complete singles to produce a list with only the first occurrence of each element. You may use one of the built-in functions remove-all or remove.

```
;; (singles alon) produces a list containing only the first occurrence
;;    of each value in alon.
;; singles: (listof Num) → (listof Num)

(define (singles alon)
  (cond
    [(empty? alon) ...]
    [else (... (first alon) ... (singles (rest alon)) ...)]))
```

Wrapping a function in another function

Sometimes it is convenient to have a recursive helper function that is "wrapped" in another function.

Consider using a wrapper function

- if you need the data in a different format, or
- if you find that in your recursive function you need more parameters than a lab or assignment question specifies.

Wrapper functions will be very useful when designing many functions that consume and process strings. The "wrapped" helper function will typically do most of the work of solving the problem.

Strings and characters

Strings are made up of zero or more character values.

In Racket, characters are a separate type.

Example character values: #\space, #\1, #\a.

Built-in functions: char<? to compare two lower-case letters, two upper-case letters, or two digits.

Built-in predicates char-upper-case?.

The course web page has more information about characters.

We will use the type name Char in contracts, as needed.

Strings and lists of characters

Although strings and lists of characters are two different types of data, we can convert back and forth between them.

The built-in function list->string converts a list of characters into a string, and string->list converts a string into a list of characters.

This allows us to have the convenience of the string representation and the power of the list representation.

We use lists of characters in labs, assignments, and exams. We will use helper functions that accept lists of characters to actually solve many string problems.

Using wrappers for string functions

One way of building a function that consumes and produces strings:

- convert the string to a list of characters,
- write a function (using the list template) that consumes and produces a list of characters, and then
- convert the list of characters to a string.

For convenience, you may write examples and tests for the wrapper only (that is, for *strings*, not *lists of characters*).

String function example

```
;; Example of Wrapped helper function
;; list-of-char-helper-fn: (listof Char) → (listof Char)
;; Note: You do not need to include examples and tests for the
;;              wrapped helper function.
(define (list-of-char-helper-fn loc) (...))

;; Example of Wrapper function
;; string-template: Str → Str
(define (string-template s)
    (list->string (list-of-char-helper-fn (string->list s))))
```

Canadianizing a string

```
;; (canadianize s) produces a string in which each o in s
;;    is replaced by ou.
;; canadianize: Str → Str
;; Examples:
(check-expect (canadianize "") "")
(check-expect (canadianize "mold") "mould")
(check-expect (canadianize "zoo") "zouou")

(define (canadianize str)
   (list->string (canadianize-list (string->list str))))
```

You will write canadianize-list in lab.

Determining portions of a total

portions produces a list of fractions of the total represented by each number in a list, or (portions (cons 3 (cons 2 empty))) would yield (cons 0.6 (cons 0.4 empty))

We can write a function total-list that computes the total of all the values in list.

For an empty list, we produce the empty list.

For a nonempty list, we cons the first item divided by the total onto (portions (rest alon)).

This algorithm fails to solve the problem because total-list is being reapplied to smaller and smaller lists.

We just want to compute the total once and then have the total go along for the ride in our calculation.

We create a function portions-with-total that takes the total along for the ride.

Now portions is a wrapper that uses total-list to determine the total for the whole list and then uses it as an argument to a function application of portions-with-total.

Nonempty lists

Sometimes a given computation only makes sense on a nonempty list – for instance, finding the maximum of a list of numbers. If the function requires that a parameter is a nonempty list, we add a **requires** section.

```
;; A nonempty list of numbers (Neln) is either:
;; * (cons Num empty)
;; * (cons Num Neln)
```

```
;; (max-list lon) produces the maximum element of lon
;; max-list: NeIn → Num

(define (max-list lon)
    ...)
```

Functions with multiple base cases

Suppose we wish to determine whether all values in a list of numbers are equal.

What should the function produce if the list is empty?

What if the list has only one item?

Goals of this module

You should be comfortable with these terms: recursive, recursion, self-referential, base case, structural recursion.

You should understand the data definitions for lists (including nonempty lists), how the template mirrors the definition, and be able to use the templates to write recursive functions consuming this type of data.

You should understand box-and-pointer and nested boxes visualizations of lists.

You should understand the additions made to the syntax of Beginning Student to handle lists, and be able to do step-by-step traces on list functions.

You should be aware of situations that require wrapper functions.

You should understand how to use (listof . . .) notation in contracts, as well as how to add restrictions on lists and list values.

Module 5: Working with recursion

Readings: HtDP, sections 11, 12, 13 (Intermezzo 2)

We can extend the idea of a self-referential definition to defining the natural numbers, which leads to the use of recursion in order to write functions that consume numbers.

Note that natural numbers can be defined non-recursively. But, our focus is on a recursive view.

Natural numbers

;; **natural number** (Nat) is either:

;; * 0

;; * 1 + Nat

The analogy to the self-referential definition of lists can be made clearer by using the built-in add1 function:

(add1 0) $\Rightarrow$ 1

(add1 (add1 0)) $\Rightarrow$ 2

(add1 (add1 (add1 0))) $\Rightarrow$ 3

```
;; A List is one of:
;; * empty
;; * (cons Any List)
```

To derive a template, we used a cond for the two cases.

We broke up the nonempty list (cons f r) using

- the selector first to extract f,
- the selector rest to extract r, and
- an application of the function on r.

```
;; A Nat is one of:
;; * 0
;; * (add1 Nat)
```

To derive a template for a natural number n, we will use a cond for the two cases.

We will break up the non-zero case n=k+1 using

- the function sub1 to extract k and
- an application of the function on k (that is, on (sub1 n)).

Comparing the templates

```
;; list-template: (listof Any) → Any
(define (list-template alist)
  (cond
    [(empty? alist) ...]
    [else (... (first alist) ... (list-template (rest alist)) ...)]))
;; nat-template: Nat → Any
(define (nat-template n)
  (cond
    [(zero? n) ...]
    [else (... n ... (nat-template (sub1 n)) ...)]))
```

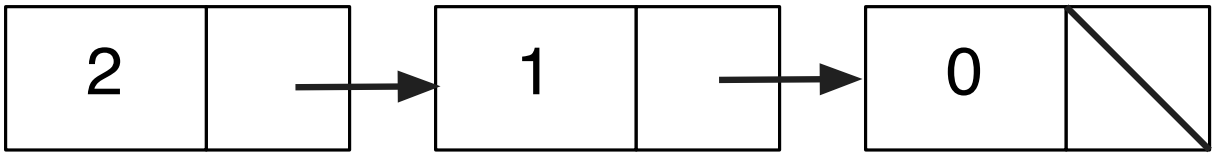
Example: producing a decreasing list

countdown consumes a natural number n and produces a decreasing list of all natural numbers less than or equal to n.

Use the data definition to derive examples.

(countdown 0) $\Rightarrow$ (cons 0 empty)

(countdown 2) $\Rightarrow$ (cons 2 (cons 1 (cons 0 empty)))



Developing countdown

Using the natural number template:

```
(define (countdown n)
  (cond
    [(zero? n) ...]
    [else (... n ... (countdown (sub1 n)) ...)]))
```

If n is 0, we produce the list containing 0.

If n is nonzero, we cons n onto the countdown list for n-1.

```
;; (countdown n) produces a decreasing list of nats starting at n
;; and ending with 0
;; countdown: Nat → (listof Nat)
;; Examples:
(check-expect (countdown 0) (cons 0 empty))
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

(define (countdown n)
  (cond [(zero? n) (cons 0 empty)]
        [else (cons n (countdown (sub1 n)))]))
```

Condensed trace of countdown

(countdown 2)

$\Rightarrow$ (cons 2 (countdown 1))

$\Rightarrow$ (cons 2 (cons 1 (countdown 0)))

$\Rightarrow$ (cons 2 (cons 1 (cons 0 empty)))

If the function countdown is applied to a negative argument, it will not terminate.

The following variation is a little more robust.

It can handle negative arguments more gracefully.

```
(define (countdown n)
  (cond
    [(<= n 0) (cons 0 empty)]
    [else (cons n (countdown (sub1 n)))]))
```

Counting down to a different base

We can generalize countdown by providing a base value as a parameter b.

This function deals with the following range of integers:

An **integer greater than or equal to** b is either

- b or
- 1 plus *an integer greater than or equal to* b.

The parameter b (for "base") has to "go along for the ride" in the recursion.

Template for an integer greater than or equal to a base

```
;; downto-template: Int Int → Any
;; requires: n >= b
(define (downto-template n b)
  (cond
    [(= n b) (... b ...)]
    [else (... n ... (downto-template (sub1 n) b)...)]))
```

The template nat-template is a special case where b is zero. Since we know the value zero, it doesn't need to be a parameter.

The function countdown-to

```
;; (countdown-to n b) produces a decreasing list of ints starting with n
;;    and ending with b.
;; countdown-to: Int Int → (listof Int)
;; requires: n >= b
(define (countdown-to n b) . . . )
   (cond
      [(<= n b) (cons b empty)]
      [else (cons n (countdown-to (sub1 n) b))]))
```

Condensed trace of countdown-to

(countdown-to 4 2)

$\Rightarrow$ (cons 4 (countdown-to 3 2))

$\Rightarrow$ (cons 4 (cons 3 (countdown-to 2 2)))

$\Rightarrow$ (cons 4 (cons 3 (cons 2 empty)))

- What is the result of (countdown-to 1 −2)?
- Of (countdown-to −4 −2)?

Going the other way

What if we want an increasing count up to an integer b?

Consider the following definition:

An **integer less than or equal to b** is either:

- b
- An *(integer less than or equal to b) minus 1*

Based on this definition, the following are integers less than or equal to $b = 14$: 14, (sub1 14), (sub1 (sub1 14)), ...

Our recursive step n=k-1 will use (add1 n).

```
;; upto-template: Int Int → Any
;; requires: n <= b
(define (upto-template n b)
  (cond
    [(>= n b) (... b ...)]
    [else (... n ... (upto-template (add1 n) b) ...)]))
```

```
;; (countup-to n b) produces an increasing list of ints starting with n
;;    and ending with b
;; countup-to: Int Int → (listof Int)
;; requires: n <= b
;; Examples:
(check-expect (countup-to 5 5) (cons 5 empty))
(check-expect (countup-to 6 8) (cons 6 (cons 7 (cons 8 empty))))

(define (countup-to n b)
  (cond
    [(>= n b) (cons b empty)]
    [else (cons n (countup-to (add1 n) b))]))
```

Condensed trace of countup-to

(countup-to 6 8)

$\Rightarrow$ (cons 6 (countup-to 7 8))

$\Rightarrow$ (cons 6 (cons 7 (countup-to 8 8)))

$\Rightarrow$ (cons 6 (cons 7 (cons 8 empty)))

Consider another variation

```
;; countup-by: Int Int Nat → (listof Int)
(define (countup-by n b inc)
    (cond [(>= n b) empty]
          [else (cons n (countup-by (+ n inc) b inc))]))

(countup-by 2 5 1)

(countup-by -10 20 7)

(countup-by 10 5 1)
```

range function

Racket has a built-in function range that generalizes countup-by, and provides a short-cut for basic counting up or down.

```
;; (range a b c) produces a list of integers from a to b, but not
;;    including b, stepping by c.
;; range: Int Int Int → (listof Int)
```

(range 4 7 1) ⇒ (cons 4 (cons 5 (cons 6 empty)))

(range 5 0 −1) ⇒ (cons 5 (cons 4 (cons 3 (cons 2 (cons 1 empty)))))

(range −4 4 3) ⇒ (cons −4 (cons −1 (cons 2 empty)))

(range 2 2 5) ⇒ empty

Many imperative programming languages offer several language constructs to do repetition:

for i = 1 to 10 do { ... }

Racket offers a construct – recursion – that is flexible enough to handle these situations and more.

We will soon see how to use Racket's abstraction capabilities to handle many common uses of recursion.

When you are learning to use recursion with integers, sometimes you will “get it backwards” and use the countdown pattern when you should be using the countup pattern, or vice-versa.

Avoid using the built-in list function reverse to fix your error. It cannot always save a computation done in the wrong order. Instead, learn to fix your mistake by starting with the right template.

Do not use reverse in your labs, assignments, or exams. You will lose many marks if you do.

Example: factorial

Suppose we wish to compute $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$, or (factorial n), for $n \geq 0$ ($0!$ is defined to be 1).

To choose between the templates, we consider what information each gives us.

Counting down: (factorial (sub1 n))

Counting up: (factorial (add1 n))

Filling in the template

```
(define (downto-template n b)
  (cond
    [(= n b) (... b ...)]
    [else (... n ... (downto-template (sub1 n) b) ...)]))

(define (factorial n)
  (cond
    [(zero? n) 1]
    [else (* n (factorial (sub1 n)))]))
```

Example: String prefixes

Suppose we want to find all the prefixes of a string, starting from the shortest prefix (the empty string) to the string itself?

For example,

(prefixes "abc") $\Rightarrow$

(cons "" (cons "a" (cons "ab" (cons "abc" empty))))

How is this a problem on natural numbers?

We need to include: (substring s 0 0), (substring s 0 1), (substring s 0 2), etc.

Example: Greatest common divisor

The greatest common divisor (gcd) of a group of positive natural numbers is the largest integer that divides evenly into each. For example, the gcd of 21, 35, 14 is 7.

Suppose we want to find the gcd of three natural numbers: n1, n2, n3.

What range of numbers should we check?

In what order should we check them?

How do we check for a common divisor?

Example: Checking for a common divisor

How do we check if a particular number is a common divisor?

```
;; (common? k n1 n2 n3) produces true if k divides evenly into
;;    n1, n2, and n3, and false otherwise.
;; common?: Nat Nat Nat Nat → Boolean
;; requires: k > 0
(define (common? k n1 n2 n3)
        (and (zero? (remainder n1 k))
             (zero? (remainder n2 k))
             (zero? (remainder n3 k))))
```

```
;; gcd-three: Nat Nat Nat → Nat
;; requires: n1, n2, n2 > 0
(define (gcd-three n1 n2 n3)
  ...)
```

How do we countdown?

gcd-three needs a recursive helper function.

A more complicated situation - Sorting

Sometimes a recursive function will use a helper function that itself is recursive.

Sorting a list of numbers provides a good example.

In CS 116, we will see several different sorting algorithms.

We will sort from lowest number to highest.

(cons 3 (cons 5 (cons 9 empty)))

A list is sorted if no number is followed by a smaller number.

Filling in the list template

```
;; (my-sort alon) produces a list containing same values as alon sorted in
;;    nondecreasing order.
;; my-sort: (listof Num) → (listof Num)
(define (my-sort alon)
   (cond
      [(empty? alon) ...]
      [else (... (first alon) ... (my-sort (rest alon)) ...)]))
```

If the list alon is empty, so is the result. Otherwise, the template suggests somehow combining the first element and the sorted version of the rest.

```
(define (my-sort alon)
  (cond
    [(empty? alon) empty]
    [else (insert (first alon) (my-sort (rest alon)))]))
```

insert is a recursive auxiliary function which consumes a number and a sorted list, and adds the number to the sorted list.

Tracing my-sort

(my-sort (cons 7 (cons 4 (cons 3 empty))))

$\Rightarrow$ (insert 7 (my-sort (cons 4 (cons 3 empty))))

$\Rightarrow$ (insert 7 (insert 4 (my-sort (cons 3 empty))))

$\Rightarrow$ (insert 7 (insert 4 (insert 3 (my-sort empty))))

$\Rightarrow$ (insert 7 (Insert 4 (insert 3 empty)))

$\Rightarrow$ (insert 7 (insert 4 (cons 3 empty)))

$\Rightarrow$ (insert 7 (cons 3 (cons 4 empty)))

$\Rightarrow$ (cons 3 (cons 4 (cons 7 empty)))

The helper function insert

We again use the list template for insert.

```
;; (insert n alon) produces the sorted (in nondecreasing order) list
;;    formed by adding the number n to the sorted list alon.
;; insert: Num (listof Num) → (listof Num)
;; requires: alon is sorted in nondecreasing order
(define (insert n alon)
  (cond
    [(empty? alon) ...]
    [ else (... (first alon) ... (insert n (rest alon)) ...)]))
```

Reasoning about insert

If alon is empty, then the result is the list containing just n.

If alon is not empty, another conditional expression is needed.

n is the first number in the resulting list if it is less than or equal to (first alon).

Otherwise, (first alon) is the first number in the resulting list, and we get the rest of the resulting list by inserting n into (rest alon).

```
(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [(<= n (first alon)) (cons n alon)]
    [ else (cons (first alon) (insert n (rest alon)))]))
```

Tracing insert

(insert 5 (cons 2 (cons 4 (cons 6 empty))))

$\Rightarrow$ (cons 2 (insert 5 (cons 4 (cons 6 empty))))

$\Rightarrow$ (cons 2 (cons 4 (insert 5 (cons 6 empty))))

$\Rightarrow$ (cons 2 (cons 4 (cons 5 (cons 6 empty))))

This is known as **insertion sort**.

List abbreviations

Now that we understand Racket lists, we can abbreviate them.

The expression

(cons exp1 (cons exp2 (. . . (cons expn empty). . .)))

can be abbreviated as

(list exp1 exp2 . . . expn)

The result of the trace we did on the last slide can be expressed as (list 2 4 5 6).

Beginning Student Racket with List Abbrevations also provides some shortcuts for accessing specific elements of lists.

(second my-list) is an abbreviation for (first (rest my-list)).

third, fourth, and so on up to eighth are also defined.

Use these sparingly to improve readability, and use list to construct long lists.

There will still remain situations when using cons is the best choice.

Note that cons and list have different results and different purposes.

We use list to construct a list of fixed size (whose length is known when we write the program). We will mostly use list in our examples and tests.

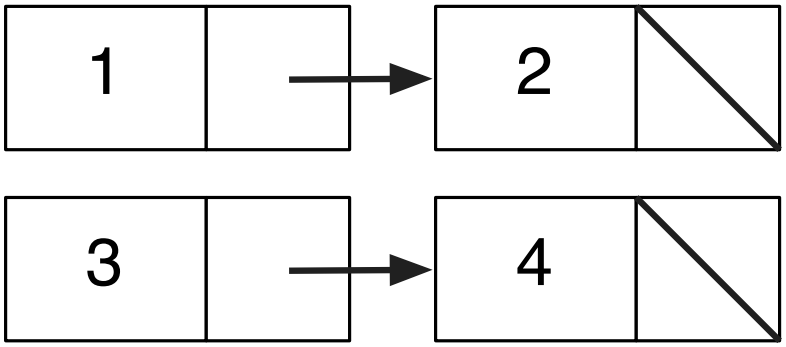
We use cons to construct a list from one new element (the first) and a list of arbitrary size (whose length is known only when the second argument to cons is evaluated during the running of the program).

Our function implementations are more likely to use cons than list.

Lists containing lists

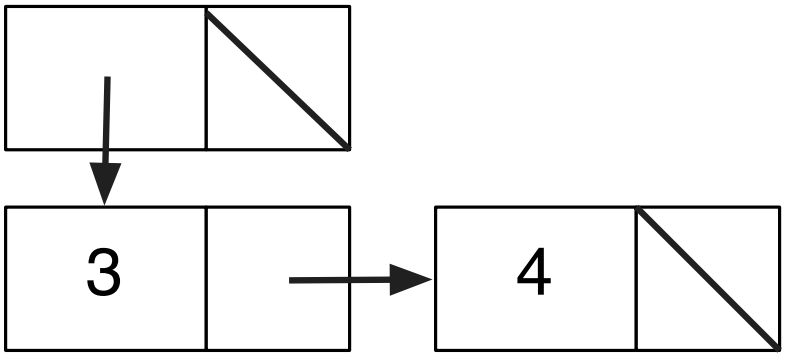
Lists can contain anything, including other lists, at which point these abbreviations can improve readability.

Here are two different two-element lists.




```
(cons 1 (cons 2 empty))
(cons 3 (cons 4 empty))
```

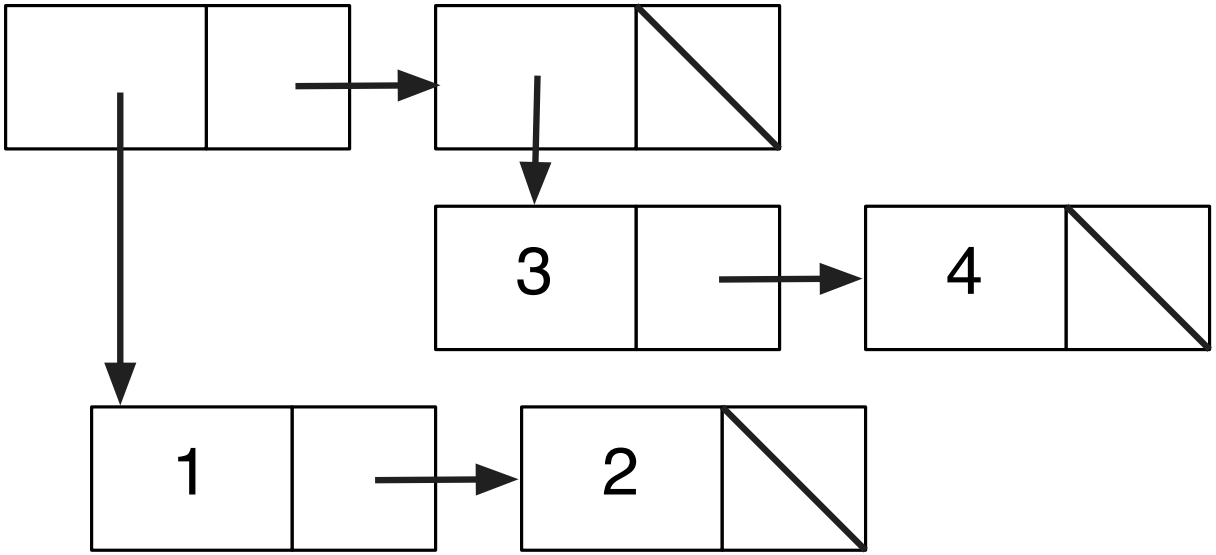
Here is a one-element list whose single element is one of the two-element lists we saw above.




```
(cons (cons 3 (cons 4 empty))
      empty)
```

We can create a two-element list by consing the other list onto this one-element list.

We can create a two-element list, each of whose elements is itself a two-element list.




```
(cons (cons 1 (cons 2 empty))
      (cons (cons 3 (cons 4 empty)) empty))
```

Expressing the list

We have several ways of expressing this list in Racket:

```
(cons (cons 1 (cons 2 empty))
      (cons (cons 3 (cons 4 empty)) empty))
```

```
(list (list 1 2) (list 3 4))
```

Many find the abbreviations more expressive and easier to use.

Using lists for related data

Aside from lists, the types we've seen so far have been very simple: a number, a string, a Boolean value.

Sometimes, it would be nice to group together related pieces of information into a single value. For example, suppose we need the name of a student, along with their grades on assignments, the midterm exam, and the final exam. We could use the four different values separately, or we could create a list with the four values inside it, and call it a Student.

For example, the following are three Student values:

```
(list "Virginia Woolf" 100 100 100)
(list "Alan Turing" 90 80 40)
(list "Anonymous" 30 55 10)
```

We can use a data definition to be precise about our new Student type.

```
;; A Student is (list Str Num Num Num)
;; where:
;; * the first item is the name of a student
;; * the second item is the assignment grade
;; * the third item is the midterm exam grade
;; * the fourth item is the final exam grade
;; * and all grades are between 0 and 100, inclusive.
```

Using (listof Student)

We can develop a template for this new type of list.

```
(define (slist-template sl)
  (cond
    [(empty? sl) ...]
    [else (...  (first (first sl)) ; name of first
                ... (second (first sl)) ; assts of first
                ... (third (first sl)) ; mid of first
                ... (fourth (first sl)) ; final of first
                ... (slist-template (rest sl))...)]))
```

Example: a function name-list which consumes a (listof Student) and produces the corresponding list of student names.

```
(define (name-list sl)
  (cond
    [(empty? sl) empty]
    [else (cons (first (first sl)) ; name of first
                (name-list (rest sl)))]))
```

This code is not very readable because the meaning of (first (first sl)) is not not clear without the data definition.

We can fix this with a few definitions.

```
(define (name x) (first x))
(define (assts x) (second x))
(define (mid x) (third x))
(define (final x) (fourth x))
(define (name-list sl)
  (cond
    [(empty? sl) empty]
    [else (cons (name (first sl))
                (name-list (rest sl)))]))
```

name-list is re-usable

If we define a **glist** as a list of two-element lists, each sublist holding name and grade, we could reuse the name-list function to produce a list of names from a glist.

We will exploit this ability to reuse code written to use “generic” lists when we discuss abstract list functions later in the course.

What happens with the following call?

```
(name-list (list (list 1 "a") (list 3 "c") (list 2 "b")))
```

Dictionaries

You know dictionaries as books or a website in which you look up a word and get a definition or a translation.

More generally, a dictionary contains a number of unique **keys**, each with an associated **value**. For example,

- Seat assignment look-up (keys=usernames, values=seats)
- Reverse telephone lookup (keys=phone numbers, values=names).

Our task is to store the set of (key,value) pairs to support the operations *lookup*, *add*, and *remove*.

Association lists

```
;; An association (As) is (list Num Str),
;; where
;; * the first item is the key,
;; * the second item is the associated value.

;; An association list (AL) is one of
;; * empty
;; * (cons As AL)
;; Note: All keys must be distinct.
```

Without these data definitions, we can write the type as (listof (list Num Str))

```
;; (lookup-al k alst) produces the value associated with k, or
;;    false if k not present.
;; lookup-al: Num AL → (anyof Str false)
(define (lookup-al k alst)
  (cond
    [(empty? alst) false]
    [(equal? k (first (first alst))) (second (first alst))]
    [else (lookup-al k (rest alst))]))
```

We will leave the add and remove functions as exercises.

This solution is simple enough that it is often used for small dictionaries.

For a large dictionary, association lists are inefficient in the case where the key is not present and the whole list must be searched.

Keeping the list in sorted order might improve some searches, but there is still a case where the whole list is searched.

In a later module, we will see how to avoid this.

Different kinds of lists

When we introduced lists in Module 4, the items they contained were not lists. These were **flat lists**.

We have just seen **lists of lists** in our example of lists containing two-element flat lists.

Later, we will see **nested lists**, in which lists may contain lists that contain lists, and so on to an arbitrary depth.

Goals of this module

You should understand the recursive definition of a natural number, and how it leads to a template for recursive functions that consume natural numbers.

You should understand how subsets of the integers greater than or equal to some bound b, or less than or equal to such a bound, can be defined recursively, and how this leads to a template for recursive functions that "count down" or "count up". You should be able to write such functions.

You should understand the principle of insertion sort, and how the functions involved can be created using the design recipe.

You should be able to use list abbreviations for lists where appropriate.

You should be able to construct and work with lists that contain lists.

You should understand the uses of fixed-size lists, and be able to write functions that consume or produce such data.

You should be able to use association lists to implement dictionaries.

Module 6: Processing two lists or numbers

Readings: HtDP, section 17.

Processing two lists simultaneously

We now move to using recursion in a complicated situation, namely writing functions that consume two lists (or two data types, each of which has a recursive definition).

Following Section 17 of the textbook, we will distinguish three different cases, and look at them in order of complexity.

One goal of this module is to learn how to choose among cases.

Case 1: a list going along for the ride

my-append consumes two lists list1 and list2, and produces the list containing all the values in list1 followed by all the values in list2.

It effectively creates a new list from list1 by replacing its empty list with list2.

Do we need to process the individual elements of list1?

What about the individual elements of list2?

```
;; list-template: (listof Any) → Any
(define (list-template alist)
  (cond [(empty? alist) ...]
        [else (... (first alist) ...
                   (list-template (rest alist)) ...)]))

;; alongforride-template: (listof Any) (listof Any) → Any
(define (alongforride-template list1 list2)
  (cond [(empty? list1) ...]
        [else (... (first list1) ...  (alongforride-template (rest list1) list2) ...)]))
```

alongforride-template is a slight variation of extra-list-info-template in M04.

The function my-append

```
;; my-append: (listof Any) (listof Any) → (listof Any)
;; Examples:
(check-expect (my-append empty (list 1 2)) (list 1 2))
(check-expect (my-append (list 1) (list 2 3)) (list 1 2 3))

(define (my-append list1 list2)
  (cond [(empty? list1) list2]
        [else (cons (first list1) (my-append (rest list1) list2))]))
```

Condensed trace of my-append

(my-append (list 1 2) (list 3 4))

$\Rightarrow$ (cons 1 (my-append (list 2) (list 3 4)))

$\Rightarrow$ (cons 1 (cons 2 (my-append empty (list 3 4))))

$\Rightarrow$ (cons 1 (cons 2 (list 3 4)))

$\Rightarrow$ (list 1 2 3 4)

Built-in append

The Racket function append is a built-in function like my-append.

Unlike our version, append may consume two or more lists.

You may use append on assignments, unless told otherwise.

Both (append (list 3) my-list) and (cons 3 my-list) produce the same list. Our style preference is to use cons rather than append when the first list has length one.

Case 2: processing in lockstep

total-value determines the total value of items sold, given prices of items and numbers of each item.

Example: the total of prices $(1\ 2\ 3)$ and numbers $(4\ 5\ 6)$ is $1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32$.

We can store the items (and numbers) in lists, so $(1\ 2\ 3)$ becomes (list 1 2 3).

Key observation: the lists have the same length and items in a given position correspond to each other.

```
;; (total-value pricelist numlist) produces the total value of items
;;    with prices in pricelist and numbers in numlist.
;; total-value: (listof Num) (listof Nat) → Num
;; requires: Values in pricelist >= 0
;;           pricelist and numlist have the same length
;; Examples:
(check-expect (total-value empty empty) 0)
(check-expect (total-value (list 2) (list 3)) 6)
(check-expect (total-value (list 2 3) (list 4 5)) 23)
```

Developing a template

pricelist is either empty or a cons, and the same is true of numlist, yielding four options of empty/non-empty lists.

However, because the two lists must be the same length, (empty? pricelist) is true if and only if (empty? numlist) is true.

Out of the four possibilities, two are invalid for proper data.

The template is thus simpler than in the general case.

Extracting the lockstep template

```
;; lockstep-template: (listof Any) (listof Any) → Any
;; requires: list1 and list2 are the same length
(define (lockstep-template list1 list2)
  (cond
    [(empty? list1) ... ]
    [else
      (... (first list1) ... (first list2) ...
           (lockstep-template (rest list1) (rest list2)) ...)]))
```

Completing total-value

```
(define (total-value pricelist numlist)
  (cond
    [(empty? pricelist) 0 ]
    [else
      (+
        (* (first pricelist) (first numlist))
        (total-value (rest pricelist) (rest numlist)))]))
```

Condensed trace of total-value

(total-value (list 2 3) (list 4 5))

$\Rightarrow$ (+ (* 2 4) (total-value (list 3) (list 5)))

$\Rightarrow$ (+ 8 (total-value (list 3) (list 5)))

$\Rightarrow$ (+ 8 (+ (* 3 5) (total-value empty empty)))

$\Rightarrow$ (+ 8 (+ 15 (total-value empty empty)))

$\Rightarrow$ (+ 8 (+ 15 0))

$\Rightarrow$ (+ 8 15)

$\Rightarrow$ 23

Case 3: processing at different rates

If the lists being consumed, list1 and list2, are of different lengths, all four possibilities for their being empty/nonempty are possible:

```
(and (empty? list1) (empty? list2))
(and (empty? list1) (cons? list2))
(and (cons? list1) (empty? list2))
(and (cons? list1) (cons? list2))
```

Exactly one of these is true for a given pair of lists, but all possibilities must be included in the template.

The template so far:

```
;; twolist-template: (listof Any) (listof Any) → Any
(define (twolist-template list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) ...]
    [(and (empty? list1) (cons? list2)) ...]
    [(and (cons? list1) (empty? list2)) ...]
    [(and (cons? list1) (cons? list2)) ...]))
```

The first situation is a **base case**.

Refining the template

```
(define (twolist-template list1 list2)
 (cond
   [(and (empty? list1) (empty? list2)) ...]
   [(and (empty? list1) (cons? list2))
     (... (first list2) ... (twolist-template empty (rest list2)) ...)]
   [(and (cons? list1) (empty? list2))
     (... (first list1) ... (twolist-template (rest list1) empty)...)]
   [(and (cons? list1) (cons? list2)) ??? ]))
```

The fourth case definitely does, but its form is unclear.

Further refinements

There are several different possible natural recursions for the last cond answer ??? :

... (first list2) ... (twolist-template list1 (rest list2)) ...

(first list1) ... (twolist-template (rest list1) list2) ...

(first list1) ... (first list2)

(twolist-template (rest list1) (rest list2)) ...

We write all of these down, realizing that not all will be used, and eliminate unnecessary ones in reasoning about any particular function.

```
(define (twolist-template list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) ...]
    [(and (empty? list1) (cons? list2))
     (... (first list2) ... (twolist-template empty (rest list2)) ...)]
    [(and (cons? list1) (empty? list2))
     (... (first list1) ... (twolist-template (rest list1) empty)...)]
    [(and (cons? list1) (cons? list2))
     (... (first list2) ... (twolist-template list1 (rest list2)) ...
          (first list1) ... (twolist-template (rest list1) list2) ...
          (first list1) ... (first list2)...
          (twolist-template (rest list1) (rest list2))...)]))
```

Example: merging two sorted lists

We wish to design a function merge that consumes two lists, each of distinct elements sorted in ascending order, and produces one sorted list containing all elements.

For example:

(merge (list 1 8 10) (list 2 4 6 12)) $\Rightarrow$ (list 1 2 4 6 8 10 12)

We need more examples to be confident of how to proceed.

(merge empty empty) ⇒ empty

(merge empty
(cons 2 empty)) ⇒ (cons 2 empty)

(merge (cons 1 (cons 3 empty))
empty) ⇒ (cons 1 (cons 3 empty))

(merge (cons 1 (cons 4 empty))
(cons 2 empty)) ⇒ (cons 1 (cons 2 (cons 4 empty)))

(merge (cons 3 (cons 4 empty))
(cons 2 empty)) ⇒ (cons 2 (cons 3 (cons 4 empty)))

Reasoning about merge

If list1 and list2 are both nonempty, what is the first element of the merged list?

It is the smaller of (first list1) and (first list2).

If (first list1) is smaller, then the rest of the answer is the result of merging (rest list1) and list2.

If (first list2) is smaller, then the rest of the answer is the result of merging list1 and (rest list2).

```
(define (merge list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) empty]
    [(and (empty? list1) (cons? list2)) list2]
    [(and (cons? list1) (empty? list2)) list1]
    [(< (first list1) (first list2))
        (cons (first list1) (merge (rest list1) list2))]
    [else
        (cons (first list2) (merge list1 (rest list2)))]))
```

Condensed trace of merge

```
(merge (cons 3 (cons 4 empty))
       (cons 2 (cons 5 (cons 6 empty))))
⇒ (cons 2 (merge (cons 3 (cons 4 empty))
                 (cons 5 (cons 6 empty))))
⇒ (cons 2 (cons 3 (merge (cons 4 empty)
                         (cons 5 (cons 6 empty)))))
⇒ (cons 2 (cons 3 (cons 4 (merge empty
                                 (cons 5 (cons 6 empty))))))
⇒ (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty)))))
```

A new data type Symbol (Sym)

Racket allow us to define and use **symbols** with meaning to us.

Symbols have a special marker to indicate that they are not function names or strings.

Syntax rule: a **symbol** starts with a single quote ’ followed by something obeying the rules for identifiers.

The symbol ’blue is a value like 5, but is more limited computationally.

Using symbols

- Symbols can be used to avoid using numbers or strings to represent a small collection of values, such as colours, names of planets, or movie genres.
- Symbols can be compared using the functions symbol=? and equal?.

```
(define favourite 'yellow)
(symbol=? favourite 'red) ⇒ false
```

Limitations: Symbols cannot be compared or processed in any other way other than checking for equality.

Symbols versus strings

Symbols

- are **atomic** (indivisible) data, like numbers
- can be compared very efficiently for equality
- should be used when only a small set of values are needed

Strings

- are **compound** data, as a string is a sequence of characters
- have lots of built-in functions
- should be used when a wide range of values are possible

```
;; (new-list p action) produces a new 2-element list from p, by
;;      reversing the two entries of p if action is 'rev, taking absolute
;;      value of each entry of p if action is 'abs, negating each entry
;;      of p if action is 'neg, and otherwise produces p.
;; new-list: (list Num Num) Sym → (list Num Num)
(define (new-list p action)
  (cond [(symbol=? action 'rev) (list (second p) (first p))]
        [(symbol=? action 'abs) (list (abs (first p)) (abs (second p)))]
        [(symbol=? action 'neg) (list (- (first p)) (- (second p)))]
        [else p]))
```

Consuming a list and a number

In Module 6, we saw how to use structural recursion on natural numbers as well as lists.

We can extend our idea for computing on two lists to computing on a list and a number, or on two numbers.

Problem: Determining if *elem* appears at least *n* times in a list.

Example: Does 'z appear at least 3 times in (list 'd 'z 'z 'h 'z 'd)?
Does 'a appear at least 2 times?

The function at-least?

```
;; (at-least? n elem lst) produces true if elem appears at
;;    least n times in lst, and false if not.
;; at-least?: Nat Any (listof Any) → Bool
;; Examples:
(check-expect (at-least? 0 'a empty) true)
(check-expect (at-least? 3 "hi" empty) false)
(check-expect (at-least? 2 'red (list 'red 'blue 'red 'green)) true)
(check-expect (at-least? 1 7 (list 5 4 0 5 3)) false)

(define (at-least? n elem lst) ...)
```

Developing the template

The recursion will involve processing the parameters n and lst. The parameter elem is ”along for the ride”, once again giving four possibilities:

```
(and (= n 0) (empty? lst))
(and (= n 0) (cons? lst))
(and (> n 0) (empty? lst))
(and (> n 0) (cons? lst))
```

Once again, exactly one of these four possibilities is true.

```
(define (list-and-nat-template n lst)
  (cond
    [(and (= n 0) (empty? lst)) ...]
    [(and (= n 0) (cons? lst))
     (... (first lst) ... (list-and-nat-template 0 (rest lst)) ...)]
    [(and (> n 0) (empty? lst))
     (... (list-and-nat-template (sub1 n) empty) ... )]
    [(and (> n 0) (cons? lst))
     (... (first lst) ... (list-and-nat-template n (rest lst)) ...
          (list-and-nat-template (sub1 n) lst) ...
          (first lst) ...
          (list-and-nat-template (sub1 n) (rest lst)) ...)]))
```

Reasoning about at-least?

What should happen when:

- n=0 and lst is empty?
- n=0 and lst is not empty?
- n>0 and lst is empty?
- n>0 and lst is not empty?

```
(define (at-least? n elem lst)
  (cond
    [(and (= n 0) (empty? lst)) true]
    [(and (= n 0) (cons? lst)) true]
    [(and (> n 0) (empty? lst)) false]
    [(and (> n 0) (cons? lst))
     (cond [(equal? elem (first lst))
            (at-least? (sub1 n) elem (rest lst))]
           [else (at-least? n elem (rest lst))])]))
```

Simplified:

```
(define (at-least? n elem lst)
  (cond [(= n 0) true]
        [(empty? lst) false]
        ;; otherwise, lst is nonempty and n > 0
        [(equal? elem (first lst))
         (at-least? (sub1 n) elem (rest lst))]
        [else
         (at-least? n elem (rest lst))]))
```

Condensed trace of at-least?

(at-least? 3 'z (list 'd 'z 'z 'h 'z 'd))

⇒ (at-least? 3 'z (list 'z 'z 'h 'z 'd))

⇒ (at-least? 2 'z (list 'z 'h 'z 'd))

⇒ (at-least? 1 'z (list 'h 'z 'd))

⇒ (at-least? 1 'z (list 'z 'd))

⇒ (at-least? 0 'z (list 'd))

⇒ true

Midpoints of pairs of point

Suppose we have a new type Point.

```
;; A Point is a (list Num Num)
;; where
;; first number is the x-coordinate
;; second number is the y-coordinate
```

Suppose we have two lists of Point values and wish to find a list of midpoints between each pair of corresponding points.

Which template should we use?

Testing list equality

We can apply the templates we have created to the question of deciding whether or not two lists of numbers are equal.

;; list=?: (listof Num) (listof Num) $\rightarrow$ Bool

Which template is most appropriate?

Applying the general two list template

```
(define (list=? list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) ...]
    [(and (empty? list1) (cons? list2))
     (... (first list2) ... (list=? empty (rest list2)) ...)]
    [(and (cons? list1) (empty? list2))
     (... (first list1) ... (list=? (rest list1) empty) ...)]
    [(and (cons? list1) (cons? list2))
     (... (first list2) ... (list=? list1 (rest list2)) ...
          (first list1) ... (list=? (rest list1) list2) ...
          (first list1) ... (first list2)... (list=? (rest list1) (rest list2))...)]))
```

Reasoning about list equality

Two empty lists are equal.

If one list is empty and the other is not, the lists are not equal.

If two nonempty lists are equal, then their first elements are equal, and their rests are equal.

The natural recursion in this case is

```
(list=? (rest list1) (rest list2))
```

The function list=?

```
(define (list=? list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) true]
    [(and (empty? list1) (cons? list2)) false]
    [(and (cons? list1) (empty? list2)) false]
    [else
      (and (= (first list1) (first list2))
           (list=? (rest list1) (rest list2)))]))
```

Another approach to the problem

Another way of viewing the problem comes from the observation that if the lists are equal, they will have the same length.

We can then use the structure of one list in our function, checking that the structure of the other list matches.

This implies the use of the lockstep template.

Here is the result of applying the lockstep template.

```
(define (list=? list1 list2)
  (cond
    [(empty? list1) ... ]
    [else
      (... (first list1) ... (first list2) ...
           (list=? (rest list1) (rest list2))...) ]))
```

Reasoning about list equality again

If the first list is empty, the answer depends on whether or not the second list is empty.

If the first list is not empty, then the following should all be true:

- the second list must be nonempty
- the first elements must be equal
- the rests must be equal

Note that the order in which conditions are checked is very important.

```
(define (list=? list1 list2)
  (cond
    [(empty? list1) (empty? list2) ]
    [else
      (and (cons? list2)
           (and (= (first list1) (first list2))
                (list=? (rest list1) (rest list2))))]))
```

Built-in list equality

As you know, Racket provides the predicate equal? which tests structural equivalence. It can compare two atomic values, two structures, or two lists. Each of the nonatomic objects being compared can contain other lists or structures.

At this point, you can see how you might write equal? if it were not already built in. It would involve testing the type of data supplied, and doing the appropriate comparison, recursively if necessary.

Goals of this module

You should understand the three approaches to designing functions that consume two lists (or a list and a number, or two numbers) and know which one is (or ones are) suitable in a given situation.

Module 7: Local and functional abstraction

Readings: HtDP, Intermezzo 3 (Section 18); Sections 19-23.

We will cover material on functional abstraction in a somewhat different order than the text.

We will only cover built-in functions that consume functions as inputs.

We move to the Intermediate teaching language with the introduction of local definitions and abstract list functions.

What is abstraction?

Abstraction consists of

- finding similarities or common aspects, and
- forgetting unimportant differences.

For a single function, differences in parameter values are forgotten, and the similarity is captured in the function body.

For multiple functions, similarity is captured in templates.

For multiple functions, further abstraction is possible.

Eating apples

```
(define (eat-apples alist)
  (cond
    [(empty? alist) empty]
    [(not (symbol=? (first alist) 'apple))
     (cons (first alist) (eat-apples (rest alist)))]
    [else (eat-apples (rest alist))]))
```

Selecting even numbers

```
(define (select-even alist)
  (cond
    [(empty? alist) empty]
    [(even? (first alist)) (cons (first alist) (select-even (rest alist)))]
    [else (select-even (rest alist))]))
```

Abstracting from these examples

Similarity: general structure (removing certain items)

Difference: predicate used to decide what to remove

Goal: form an **abstract list function** that consumes the predicate.

Functions are *first-class* values in the Intermediate Student Language.

First-class values can be bound to constants, put in lists, consumed as arguments, and produced as results.

The abstract list function filter

The following is a possible implementation of the built-in function filter. You do not need to write this yourself.

```
(define (my-filter pred alist)
  (cond
    [(empty? alist) empty]
    [(pred (first alist))
             (cons (first alist) (my-filter pred (rest alist)))]
    [else (my-filter pred (rest alist))]))
```

Tracing filter

(filter even? (list 6 7 8))

⇒ (cons 6 (filter even? (list 7 8)))

⇒ (cons 6 (filter even? (list 8)))

⇒ (cons 6 (cons 8 (filter even? empty)))

⇒ (cons 6 (cons 8 empty))

The **abstract list function** filter performs the general operation of selecting items from lists.

Racket provides such functions to apply common patterns of structural recursion.

Using filter

```
(define (select-even alist) (filter even? alist))
(define (symbol-not-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples alist) (filter symbol-not-apple? alist))
```

filter consumes a predicate specifying which elements to include in the new list. The predicate must consume only one parameter while producing a Boolean. The elements in the list are the same type as consumed by the predicate. The following contract shows the relationships between the parameters, where X can be any type.

;; filter: (X → Bool) (listof X) → (listof X)

Advantages of functional abstraction

Functional abstraction is the process of creating abstract functions such as filter.

It reduces code size.

It avoids cut-and-paste.

Bugs can be fixed in one place instead of many.

Improving one functional abstraction improves many applications.

Additions to syntax and semantics with Intermediate

A value can now be a primitive operation or a function.

Names of functions are now expressions.

Abstracting from examples

```
(define (negate-list numlist)
  (cond
    [(empty? numlist) empty]
    [else (cons (— (first numlist)) (negate-list (rest numlist)))]

(define (name-list slist)
  (cond
    [(empty? slist) empty]
    [else (cons (first (first slist))
                (name-list (rest slist)))]))
```

The abstract list function map

The following is a possible implementation of the built-in function map. You do not need to write it yourself.

```
(define (my-map f alist)
  (cond
    [(empty? alist) empty]
    [else (cons (f (first alist)) (my-map f (rest alist)))]))
```

For this and other built-in abstract list functions, see the table on page 313 of the text (Figure 57 in Section 21.2).

Tracing map

(map sqr (list 3 6 5))

⇒ (cons (sqr 3) (map sqr (list 6 5)))

⇒ (cons 9 (map sqr (list 6 5)))

⇒ (cons 9 (cons (sqr 6) (map sqr (list 5))))

⇒ (cons 9 (cons 36 (map sqr (list 5))))

⇒ (cons 9 (cons 36 (cons (sqr 5) (map sqr empty))))

⇒ (cons 9 (cons 36 (cons 25 (map sqr empty))))

⇒ (cons 9 (cons 36 (cons 25 empty)))

The abstract list function map performs the operation of transforming a list element-by-element into another list of the same length.

(map f (list x1 x2 . . . xn)) has the same effect as

(list (f x1) (f x2) . . . (f xn)).

Short examples using map:

```
(define (negate-list numlist) (map - numlist))
(define (name-list slist) (map first slist))
```

The function consumed by map must be a one-parameter function where the type of the parameter is the same as the type of the elements of the list.

Its contract can be written as follows, where X and Y can be any types.

;; map: (X $\rightarrow$ Y) (listof X) $\rightarrow$ (listof Y)

Abstracting from examples

```
(define (first-evens-from k n)
    (cond [(= n k) empty]
          [else (cons (* k 2) (first-evens-from (add1 k) n))]))
(define (first-evens n) (first-evens-from 0 n))

(define (points-from k n a b)
   (cond [(= n k) empty]
         [else (cons (list k (+ (* k a) b))
                     (points-from (add1 k) n a b))]))
(define (first-points n a b) (points-from 0 n a b))
```

These functions create a list by applying an operation to 0,1,...,n-1.

The abstract list function build-list

The following are two possible implementations of the built-in function build-list. You do not need to write these.

```
(define (build-from q n fn)
  (cond [(= n q) empty]
        [else (cons (fn q) (build-from (add1 q) n fn))]))
(define (my-build-list n fn) (build-from 0 n fn))

(define (another-build-list n fn)
  (map fn (range 0 n 1)))
```

Tracing build-list

(build-list 3 add1)

⇒ (build-from 0 3 add1)

⇒ (cons 1 (build-from 1 3 add1))

⇒ (cons 1 (cons 2 (build-from 2 3 add1)))

⇒ (cons 1 (cons 2 (cons 3 (build-from 3 3 add1))))

⇒ (cons 1 (cons 2 (cons 3 empty)))

The abstract list function build-list creates a list of length n by applying the function fn to the values 0,1,2,. . . ,n-1.

(build-list n fn) creates (list (fn 0) (fn 1) . . . (fn (− n 1)))

Short definitions using build-list

```
(define (first-squares n) (build-list n sqr))
(define (make-odd k) (+ (* 2 k) 1))
(define (first-odds n) (build-list n make-odd))
```

The function consumed by build-list must be a one-parameter function that consumes a Nat. The type in the new list is the type produced by the consumed function.

Again, the contract can be used to show the relationship between the parameters to build-list.

;; build-list: Nat (Nat $\rightarrow$ X) $\rightarrow$ (listof X)

Abstracting from examples

```
(define (product-of-numbers alist)
  (cond
    [(empty? alist) 1]
    [else (* (first alist) (product-of-numbers (rest alist)))]))

;; requires: Strings in alist are not empty
(define (concat-firsts alist)
  (cond
    [(empty? alist) " "]
    [else (string-append (substring (first alist) 0 1)
                         (concat-firsts (rest alist)))]))
```

```
(define (list-template alist)
  (cond
    [(empty? alist) ...]
    [else (... (first alist) ... (list-template (rest alist)) ...)]))
```

To fill in the template:

Replace the first ellipsis by a base value.

Combine (first alist) and the result of a recursive call on (rest alist).

Parameters for the abstract list function: base value and combining function.

The abstract list function foldr

The following is a possible implementation of foldr. You do not need to write this.

```
(define (my-foldr combine base alist)
  (cond
    [(empty? alist) base]
    [else (combine
           (first alist)
           (my-foldr combine base (rest alist)))]))
```

Tracing foldr

(foldr f 0 (list 3 6 5))

$\Rightarrow$ (f 3 (foldr f 0 (list 6 5)))

$\Rightarrow$ (f 3 (f 6 (foldr f 0 (list 5)))

$\Rightarrow$ (f 3 (f 6 (f 5 (foldr f 0 empty)))

$\Rightarrow$ (f 3 (f 6 (f 5 0))) $\Rightarrow$ . . .

Intuitively, the effect of the application

(foldr f b (list x1 x2 . . . xn)) is to compute the value of the expression

(f x1 (f x2 (. . . (f xn b) . . .))).

foldr is short for “fold right”.

It can be viewed as “folding” a list using the provided combine function, starting from the right-hand end of the list.

It can be used to implement map, filter, and other abstract list functions.

Using foldr

```
(define (product-of-numbers alist) (foldr * 1 alist))
```

If alist is (list x1 x2 ... xn), then by our intuitive explanation of foldr, the expression (foldr * 1 alist) reduces to

```
(* x1 (* x2 (* ... (* xn 1) ...)))
```

Thus foldr does all the work of the template for processing lists in the case of product-of-numbers.

The contract for foldr, for any types X and Y:

;; foldr: (X Y $\rightarrow$ Y) Y (listof X) $\rightarrow$ Y

The combine function provided to foldr consumes two parameters:

- an item (of type X) in the list that foldr consumes and
- the result (of type Y) of applying foldr to the rest of the list.

In product-of-numbers, the $*$ function multiplies an element with the product of the rest of the list.

How does concat-firsts use these two parameters?

```
(define (combine-nonempty s t)
  (cond
    [(string=? "" s) t]
    [else (string-append (substring s 0 1) t)]))

(define (concat-firsts alist)
  (foldr combine-nonempty "" alist))
```

Consider the function my-length that calculates the length of a list:

```
(define (my-length alist)
    (cond [(empty? alist) 0]
          [else (+ 1 (my-length (rest alist)))]))
```

For my-length, the first element of the list contributes 1 to the count; its actual value is irrelevant.

Thus the function provided to foldr in this case can ignore the value of the first parameter, and just add 1 to the result on the rest of the list.

```
(define (inc x y)
  (add1 y))
(define (my-length alist)
  (foldr inc 0 alist))
```

Using foldr to produce lists

The functions we provide to foldr can also produce cons expressions, since these are also values.

Example: using foldr for negate-list.

How do we combine the following to get the new list?

- an item in the list and
- the result of the recursive call on the rest of the list?

neg-combine takes the element, negates it, and conses it onto the result of the recursive call.

```
;; neg-combine: Num (listof Num) → (listof Num)
(define (neg-combine item result-on-rest)
  (cons (- item) result-on-rest))

;; negate-list: (listof Num) → (listof Num)
(define (negate-list alist)
    (foldr neg-combine empty alist))
```

Boolean functions and foldr

To check whether a predicate p produces true for every element in a list alist, we might be tempted to try:

```
(foldr and true (map p alist))
```

Problem: and is not a function, but a special form, and this produces an error.

Solution: Racket provides andmap, which can be used like this:

```
(andmap p alist)
```

For the same reason, ormap is provided.

Imperative languages, which tend to provide inadequate support for recursion, usually provide looping constructs such as “while” and “for” to perform repetitive actions on data.

Abstract list functions cover many of the common uses of such looping constructs.

Our implementation of these functions is not difficult to understand, and we can write more if needed, but the set of looping constructs in a conventional language is fixed.

Anything that can be done with the list template can be done using foldr, without explicit recursion.

Does that mean that the list template is obsolete?

No. Experienced Racket programmers still use the list template, for reasons of readability and maintainability.

Abstract list functions should be used judiciously, to replace relatively simple uses of recursion.

Another useful function: sort

;; (sort lst less-than?) produces an ordered version of lst,

;; based on the order determined by less-than?

;; sort: (listof X) (X X $\rightarrow$ Bool) $\rightarrow$ (listof X)

A simple example:

(sort (list 4 8 6 2 10) $>$) $\Rightarrow$ (list 10 8 6 4 2)

A more complicated example:

```
(define (str-len<? s1 s2)
  (or (< (string-length s1) (string-length s2))
      (and (= (string-length s1) (string-length s2))
           (string<? s1 s2))))

(sort (list "horse" "owl" "elephant" "cat") str-len<?)
⇒ (list "cat" "owl" "horse" "elephant")
```

Suppose we want to use abstract list functions to solve:

Write a function multiples-of that consumes a positive integer, n, and a list of integers, ilist, and produces a new list containing only those values in ilist which are multiples of n.

Our attempt:

```
(define (is-mult? m)
    (zero? (remainder m n)))
(define (multiples-of n ilist)
    (filter is-mult? ilist))
```

fails. Why?

The helper function is-mult? needs the value of n but to be used by filter, it can only accept one parameter - an element of the list.

However, n only exists within the body of multiples-of.

Can we define is-mult? inside multiples-of?

Yes, but we need a new construct: local.

Local definitions

The functions and special forms we've seen so far can be arbitrarily nested – except define and check-expect.

So far, definitions have to be made "at the top level", outside any expression.

The Intermediate language provides the special form local, which contains a series of local definitions plus an expression using them, of the form

(local (def1 . . . defn) exp)

What use is this?

```
(define (multiples-of n ilist)
  (local
    [;; (is-mult? m) produces true if m is a multiple of n,
     ;;    and false otherwise.
     ;; is-mult?: Int → Bool
     (define (is-mult? m)
       (zero? (remainder m n)))]

    (filter is-mult? ilist)))
```

Note: Provide **purpose, contract, and requirements** for local helper functions.

Another example

Recall the function swap-parts from Module 2.

The function used three helper functions.

```
(define (mid num)
  (quotient num 2))
(define (front-part mystring)
  (substring mystring 0 (mid (string-length mystring))))
(define (back-part mystring)
  (substring mystring (mid (string-length mystring))))
```

The helper function mid is a helper function of the helper functions front-part and back-part.

```
(define (swap-parts mystring)
  (string-append (back-part mystring) (front-part mystring)))
```

Our solution was perfectly acceptable.

However, repeated applications, such as

(mid (string-length mystring)),

make it a bit hard to read.

It would be nice to replace repeated applications by a constant.

The special form local allows us to define a constant or a function within another function.

```
(define (swap-parts mystring)
  (local
    ((define mid (quotient (string-length mystring) 2))
     (define front (substring mystring 0 mid))
     (define back (substring mystring mid)))
    (string-append back front)))
```

Note: mid, front, and back are constants, not functions, so they have no contracts.

Like cond, local results in double parentheses.

Optional: use square brackets to improve readability.

```
(define (swap-parts mystring)
  (local
    [(define mid (quotient (string-length mystring) 2))
     (define front (substring mystring 0 mid))
     (define back (substring mystring mid))]
    (string-append back front)))
```

Re-using names with local

We have reused names before:

- n is bound to a value by define
- n is the name of a parameter

```
(define n 10)
(define (myfun n) (+ 2 n))
(myfun 6)
```

The function application (myfun 6) produces 8, not 12, due to the substitution rules for function application.

A define within a local expression may rebind a name that has already been bound to another value or expression.

```
(define (my-fun n)
  (local
    [(define (local-fun n)
       (* n 10))]
    (+ n (local-fun n))))
```

The substitution rules for local must handle this.

The semantics of local [OPTIONAL]

Key ideas:

- Create a new, unique name for each identifier defined in a local definition (e.g. mid becomes mid_0).
- Bind the new name to the value.
- Substitute the new name for the old name everywhere in the expression.
- Move all of the local definitions to the top level, evaluate, and continue.

Evaluating swap-parts

```
(swap-parts "aside")
⇒ (local [(define mid (quotient (string-length "aside") 2))
          (define front (substring "aside" 0 mid))
          (define back (substring "aside" mid))]
    (string-append back front))
 ⇒ (define mid_0 (quotient (string-length "aside") 2))
   (define front_0 (substring "aside" 0 mid_0))
   (define back_0 (substring "aside" mid_0))
   (string-append back_0 front_0)
```

```
⇒ (define mid_0 (quotient 5 2))
    (define front_0 (substring "aside" 0 mid_0))
    (define back_0 (substring "aside" mid_0))
    (string-append back_0 front_0)
⇒ (define mid_0 2)
    (define front_0 (substring "aside" 0 mid_0))
    (define back_0 (substring "aside" mid_0 ))
    (string-append back_0 front_0)
⇒ (define mid_0 2)
    (define front_0 (substring "aside" 0 2)
    (define back_0 (substring "aside" mid_0))
    (string-append back_0 front_0)
```

⇒ (define mid_0 2)
(define front_0 "as")
(define back_0 (substring "aside" mid_0))
(string-append back_0 front_0)
⇒ (define mid_0 2)
(define front_0 "as")
(define back_0 (substring "aside" 2))
(string-append back_0 front_0)
⇒ (define mid_0 2)
(define front_0 "as")
(define back_0 "ide")
(string-append back_0 front_0)

⇒ (define mid_0 2)

(define front_0 "as")

(define back_0 "ide")

(string-append "ide" front_0)

⇒ (define mid_0 2)

(define front_0 "as")

(define back_0 "ide")

(string-append "ide" "as")

⇒ "ideas"

Substitution rule

An expression of the form

(local [(define x1 exp1) . . . (define xn expn)] bodyexp) is handled as follows:

x1 is replaced with a **fresh** identifier (call it x1_0) everywhere in exp1 through expn and bodyexp.

x2 is replaced with x2_0 everywhere in exp1 through expn and bodyexp.

This process is repeated with x3 through xn.

Then, all the definitions are lifted out to the top level, yielding

```
(define x1_0 exp1)
(define x2_0 exp2)
...
(define xn_0 expn)
bodyexp
```

where the expressions have been modified to use the new names.

Read Intermezzo 3 (Section 18).

Nested local expressions

It isn't always possible to define local at the beginning of the function definition, because the definition might make assumptions that are only true in part of the code.

A typical example is that of using a list function, like first or rest, which must consume a nonempty list.

When there is one local definition that can be used throughout and one not, we end up with nested local expressions.

While possible, they are not used that often.

Using local for common subexpressions

A subexpression used twice within a function body always yields the same value.

Using local to give the reused subexpression a name improves the readability of the code.

In the following example, the function eat-apples removes all occurrences of the symbol 'apple from a list of symbols.

The subexpression (eat-apples (rest alist)) occurs twice in the code.

The function eat-apples

```
(define (eat-apples alist)
  (cond
    [(empty? alist) empty]
    [(not (symbol=? (first alist) 'apple))
          (cons (first alist) (eat-apples (rest alist)))]
    [else
           (eat-apples (rest alist))]))
```

Using local in eat-apples

```
(define (eat-apples alist)
  (cond
    [(empty? alist) empty]
    [else
      (local [(define ate-rest (eat-apples (rest alist)))]
        (cond
          [(not (symbol=? (first alist) 'apple))
           (cons (first alist) ate-rest)]
          [else ate-rest]))]))
```

In the function eat-apples, the subexpression

(eat-apples (rest alist))

appears in two different answers of the same cond expression, so only one of them will ever be evaluated.

In the next example, the subexpression

(list-max (rest alon))

appears twice. The first appearance is always evaluated, and sometimes both are. In this case, using local is more efficient, as well as more readable.

The function list-max

```
;; (list-max alon) produces maximum in alon.
;; list-max: (listof Num) → Num
;; requires: alon is nonempty
(define (list-max alon)
  (cond
    [(empty? (rest alon))  (first alon)]
    [else
     (cond
       [(> (first alon) (list-max (rest alon)))  (first alon)]
       [else  (list-max (rest alon))])]))
```

Using local in list-max

```
(define (list-max2 alon)
  (cond
    [(empty? (rest alon))  (first alon)]
    [else
     (local [(define max-rest (list-max2 (rest alon)))]
       (cond
         [(> (first alon) max-rest)  (first alon)]
         [else  max-rest]))]))
```

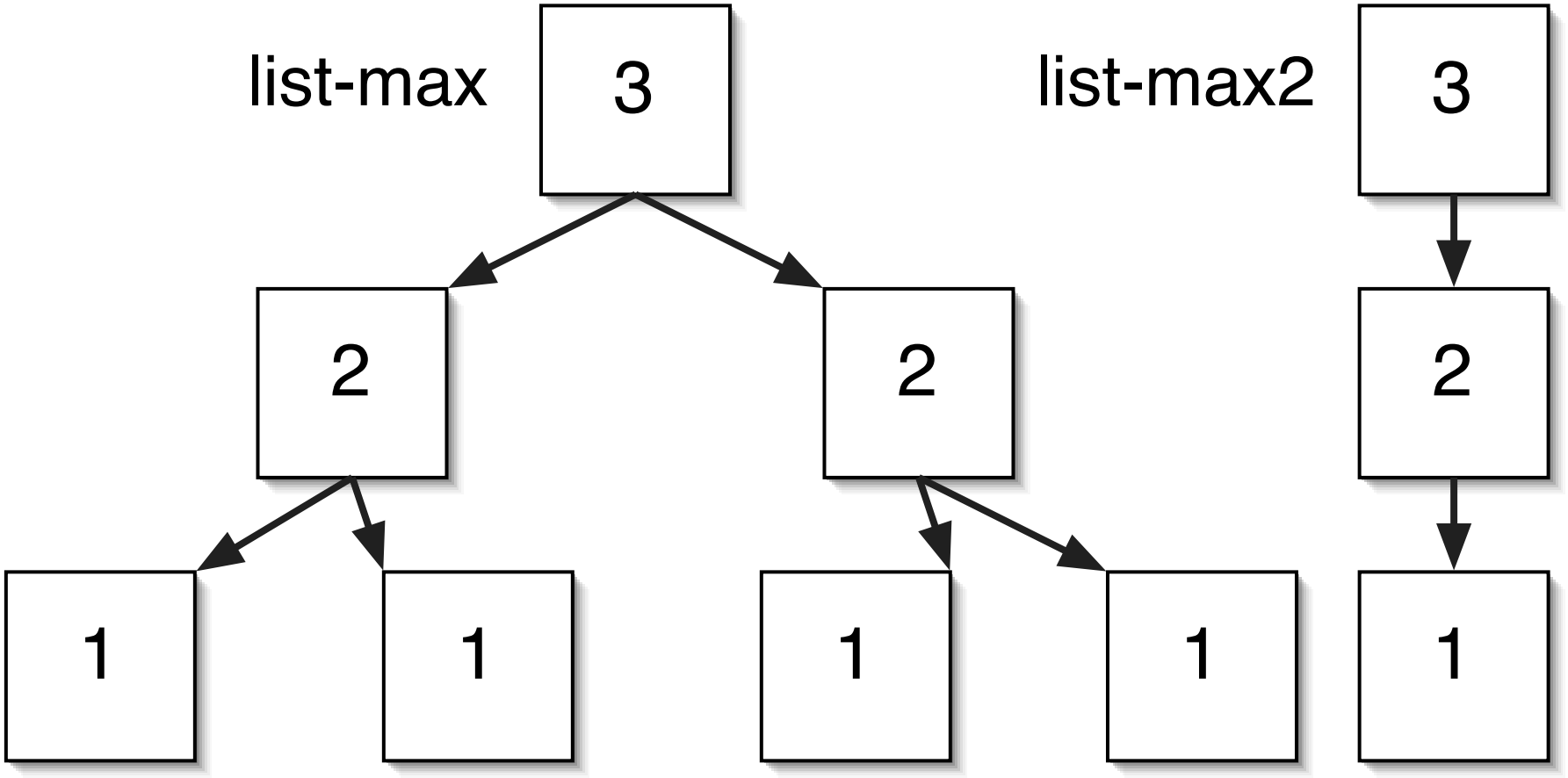
Using local can improve efficiency

You might expect that the first version does no more than twice the work of the second version.

But the first version may make two recursive calls, and each of them may make two, and so on.

If we run each version on an increasing list of three numbers, and draw a box for each call containing the length of its argument, we can see how the work adds up.

Tracing versions of list-max



Using local for smaller tasks

Sometimes we choose to use local in order to name subexpressions mnemonically to make the code more readable, even if they are not reused.

This may make the code longer, but may improve clarity.

Recall our function to compute the distance between two points.

```
;; distance: (list Num Num) (list Num Num) → Num
(define (distance point1 point2)
  (sqrt (+ (sqr (- (first point1) (first point2)))
           (sqr (- (second point1) (second point2))))))

(define (distance point1 point2)
  (local [(define delta-x (- (first point1) (first point2)))
          (define delta-y (- (second point1) (second point2)))
          (define sqrsum (+ (sqr delta-x) (sqr delta-y)))]
    (sqrt sqrsum)))
```

Using local for encapsulation

Encapsulation is the process of grouping things together in a "capsule".

Encapsulation can also be used to hide information. Here the local bindings are not visible (have no effect) outside the local expression.

We can bind names to functions as well as values in a local definition.

Evaluating the local expression creates new, unique names for the functions just as for the values.

This is known as **behaviour** encapsulation.

Behaviour encapsulation allows us to move helper functions within the function that uses them, so they are invisible outside the function.

```
(define (my-sort alon)
  (cond
    [(empty? alon) empty]
    [ else (insert (first alon) (my-sort (rest alon)))]))

(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [ else (cond
             [(<= n (first alon)) (cons n alon)]
             [ else (cons (first alon) (insert n (rest alon)))])]))
```

```
(define (my-sort alon)
  (local [;; (insert n alon) produces sorted list including n and all
          ;;    values in alon
          ;; insert: Num (listof Num) → (listof Num)
          ;; requires: alon is sorted in nondecreasing order
          (define (insert n alon)
            (cond
              [(empty? alon) (cons n empty)]
              [ else (cond
                [(<= n (first alon)) (cons n alon)]
                [ else (cons (first alon) (insert n (rest alon)))])]))] ...))
```

```
;; Body of my-sort

(cond
   [(empty? alon) empty]
   [else (insert (first alon) (my-sort (rest alon)))]))
```

Note that a full design recipe is not needed for local helper functions on assignment submissions.

You should still develop examples and tests for helper functions, and test them outside local expressions if possible.

Once you have confidence that they work, you can move them into a local expression, and delete the examples and tests.

A contract and purpose are still required (omitted in these slides for space reasons).

Another example

Use abstract list functions and local to write a function shorter-than-avg that consumes a list of strings, slist, and produces a list of all those strings in slist that are shorter than the average length of all strings in slist.

For example,

(shorter-than-avg (list "abc" "" "12345" "b")) ⇒ (list "" "b")

(shorter-than-avg empty) ⇒ empty

“Single use” helper functions

Consider the following:

```
(define (longer-strings los n)
    (local
        [(define (longer? s)
            (> (string-length s) n))]
        (filter longer? los)))
```

The function longer? is defined so it can be passed as a parameter to filter. It only exists inside longer-strings.

There is a Racket feature called lambda which allows us to simplify the definition of longer-strings.

lambda

- Creates a function value without assigning it a name.
- The function can consume any number of parameters.
- For example, (lambda (x) (+ 1 (sqr x))) is a function that consumes one parameter, and produces its square plus 1. More generally

```
(lambda
   (p1 p2 ... pN)
   body-of-function)
```

- lambda expressions cannot be simplified further.

Using lambda

Requires new Racket level: Intermediate student with lambda

```
;; (first-points n a b) produces a line containing the
;;        points (list x y)
;;        where x = 0,1,...,n-1, and y = a*x+b.

(define (first-points n a b)
    (build-list n
         (lambda (x) (list x (+ (* a x) b)))))
```

Defining functions with lambda

```
(define (double k)
    (* k 2))
```

can be rewritten using lambda as

```
(define double
   (lambda (k) (* k 2)))
```

The two definitions are equivalent and define a function called double.

Evaluating a lambda expression

Recall that a function application has the form (f a1 a2 . . . an), where f is the name of a built-in or user-defined function.

Now, f just needs to be a function value, so it can be a lambda expression. For example,

((lambda (x) (∗ 2 x)) 4) ⇒ (∗ 2 4) ⇒ 8

((lambda (s t) (+ (string-length s) t)) "hello" 10)

⇒ (+ (string-length "hello") 10)

⇒ (+ 5 10)

⇒ 15

When to use lambda?

Use lambda when the function is

- single use
- reasonably short (2-3 lines)

Complete count-starters that consumes a list of non-empty Strings (called phrases) and a String of length one (called start) and produces the number of Strings in phrases that start with start.

Two solutions - which do you prefer?

```
(define (count-starters phrases start)
  (foldr
    (lambda (s count)
      (+ count
        (cond [(string=? (substring s 0 1) start) 1]
              [else 0]))) 0 phrases))

(define (alt-count-starters phrases start)
  (length (filter (lambda (s) (string=? (substring s 0 1) start))
                  phrases)))
```

Goals of this module

You should understand the idea of encapsulation of local helper functions.

You should be familiar with map, filter, foldr, build-list, and sort, and understand how they abstract common recursive patterns, and be able to use them to write code.

You should understand the idea of functions as first-class values, and how they can be supplied as arguments.

You should understand how to do step-by-step evaluation of programs written in the Intermediate language that make use of functions as values.

You should understand the syntax, informal semantics, and formal substitution semantics for the local special form.

You should be able to use local to avoid repetition of common subexpressions, to improve readability of expressions, and to improve efficiency of code.

You should be able to match the use of any constant or function name in a program to the binding to which it refers.

You should be able to understand and write programs that use lambda expressions in place of named functions.