

Welcome to CS 135 (Winter 2017)

Instructor: Victoria Sakhnini

Web page (main information source):

<http://www.student.cs.uwaterloo.ca/~cs135/>

Other course personnel: ISAs (Instructional Support Assistants), IAs (Instructional Apprentices), ISC (Instructional Support Coordinator): see website for details

Lectures: 2 sections on Tuesdays and Thursdays

Tutorials: 3 sections on Fridays

Computer labs: MC 3003, 3004, 3005, 3027, 2062, 2063

Textbook: “How to Design Programs” (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi (<http://www.htdp.org>)

Presentation handouts: available on Web page **and** as printed coursepack from media.doc (MC 2018)

Marking Scheme: 20% assignments (roughly weekly), 25% midterm, 5% participation, 50% final

⇒ **You must pass both assignments and weighted exams in order to pass the course.**

Software: DrRacket v6.7 (<http://racket-lang.org>)

Class participation mark

- Based on use of “clickers” (purchase at Bookstore, register as part of Assignment 0)
- Purpose: to encourage active learning and provide real-time feedback
- Several multiple-choice questions during each lecture
- Marks for answering (more for correct answer)
- Best 75% over whole term used for 5% of final grade

CS 135 Survival Guide

- Available on the course Web page
- Read it as soon as possible
- You must do your own work in this course.
- Completing assignments is the key to success.
- Policy 71 - Student Discipline: plagiarism, sharing assignments, etc.

Intellectual Property

The teaching material used in CS 135 are the property of its authors.

This includes:

- Lecture slides and instructor written notes
- Assignment specifications and solutions
- Tutorial slides and notes
- Examinations and solutions

Sharing this material without the IP owner's permission is a violation of IP rights.

More suggestions for success

- Keep up with the readings (keep ahead if possible).
- Take notes in lecture.
- Start assignments early.
- Get help early.
- Follow our advice on approaches to writing programs (e.g. design recipe, templates).

- Keep on top of your workload.
- Visit office hours to get help.
- Integrate exam study into your weekly routine.
- Go beyond the minimum required (e.g. do extra exercises).
- Maintain a “big picture” perspective: look beyond the immediate task or topic.
- Go over your assignments and exams; learn from your mistakes.
- Read your mail sent to your UW email account.

Programming language design

Imperative: based on frequent changes to data

- Examples: machine language, Java, C++, Turing, VB

Functional: based on the computation of new values rather than the transformation of old ones.

- Examples: Excel formulas, LISP, ML, Haskell, Erlang, F#, Mathematica, XSLT, Clojure.
- More closely connected to mathematics
- Easier to design and reason about programs

Racket

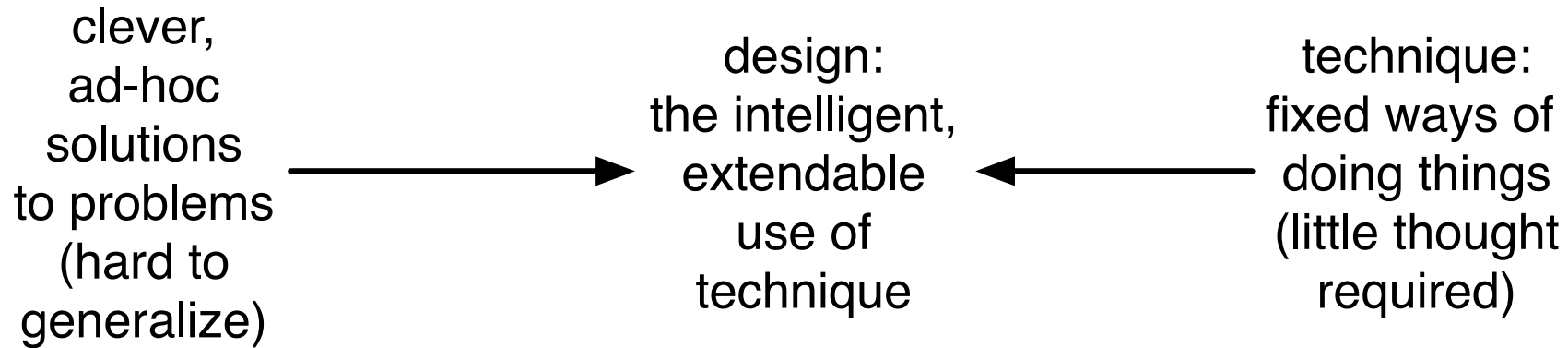
- a functional programming language
- minimal but powerful syntax
- small toolbox with ability to construct additional required tools
- interactive evaluator
- used in education and research since 1975
- a dialect of Scheme

Functional and imperative programming share many concepts. However, they require you to think differently about your programs. If you have had experience with imperative programming, you may find it difficult to adjust initially. By the end of CS 136, you will be able to express computations in both these styles, and understand their advantages and disadvantages.

Ideas behind the course

- An introduction to basic concepts in computer science
- Not a course in programming in Racket
- Teaching languages used are a small subset of standard Racket (plus some additional features)
- CS 136 continues with Racket and introduces the C language (imperative, industrially-relevant)

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, neglecting 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)

Functions in mathematics

A function generalizes similar expressions.

$$3^2 + 4(3) + 2$$

$$6^2 + 4(6) + 2$$

$$7^2 + 4(7) + 2$$

These are generalized by the function

$$f(x) = x^2 + 4x + 2.$$

Functions in mathematics

Definition: $f(x) = x^2$, $g(x, y) = x + y$

These definitions consist of:

- the name of the function (e.g. g)
- its **parameters** (e.g. x, y)
- an algebraic expression using the parameters

An **application** of a function: $g(1, 3)$

An application supplies **arguments** for the parameters, which are substituted into the algebraic expression.

Example: $g(1, 3) = 1 + 3 = 4$

The arguments supplied may themselves be applications.

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

We **evaluate** each of the arguments to yield values.

Evaluation by **substitution**:

$$g(g(1, 3), f(2)) = g(4, f(2)) = g(4, 4) = 8$$

Note the left-to-right, innermost ordering chosen.

We now enforce this.

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

Mathematically, this is a valid substitution.

We now disallow this substitution.

We insist that when we have a choice of possible substitutions, we take the **leftmost** one.

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

Mathematically, this is a valid substitution.

We now disallow this substitution.

We insist that we only apply a function when all of its arguments are values (not general expressions).

Now, for any expression:

- there is at most one choice of substitution;
- the computed final result is the same as for other choices.

The use of parentheses: ordering

In arithmetic expressions, we often place operators between their operands.

Example: $3 - 2 + 4/5$.

We have some rules (division before addition, left to right) to specify order of operation.

Sometimes these do not suffice, and parentheses are required.

Example: $(6 - 4)/(5 + 7)$.

The use of parentheses: functions

If we treat infix operators ($+$, $-$, etc.) like functions, we don't need parentheses to specify order of operations:

Example: $3 - 2$ becomes $-(3, 2)$

Example: $(6 - 4) / (5 + 7)$ becomes $/(-(6, 4), +(5, 7))$

The substitution rules we developed for functions now work uniformly for functions and operators.

Parentheses now have only one use: function application.

The use of parentheses: functions

Racket writes its functions slightly differently: the function name moves *inside* the parentheses, and the commas are changed to spaces.

Example: $g(1, 3)$ becomes `(g 1 3)`

Example: $(6 - 4) / (5 + 7)$ becomes `(/ (- 6 4) (+ 5 7))`

These are valid Racket expressions (once `g` is defined).

Functions and mathematical operations are treated exactly the same way in Racket.

Expressions in Racket

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

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

Extra parentheses are harmless in arithmetic expressions.

They are harmful in Racket.

Only use parentheses when necessary (to signal a function application or some other Racket syntax).

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)

Setting the Language in DrRacket

CS 135 will progress through the Teaching Languages starting with *Beginning Student*.

- 1. Under the *Language* tab, select *Choose Language ...*
- 2. Select *Beginning Student* under *Teaching Languages*
- 3. Click the *Show Details* button in the bottom left
- 4. Under *Constant Style*, select *true false empty*

Remember to follow steps 3 and 4 each time you change the language.

Note about Constant Style

In the DrRacket documentation, you will see `#true`, `#false`, `'()` instead of `true`, `false`, `empty`, respectively.

In CS 135 exams and stepper questions you must use `true`, `false`, `empty`.

Numbers in Racket

- Integers in Racket are unbounded.
- Rational numbers are represented exactly.
- Expressions whose values are not rational numbers are flagged as being **inexact**.

Example: `(sqrt 2)` evaluates to `#i1.414213562370951`.

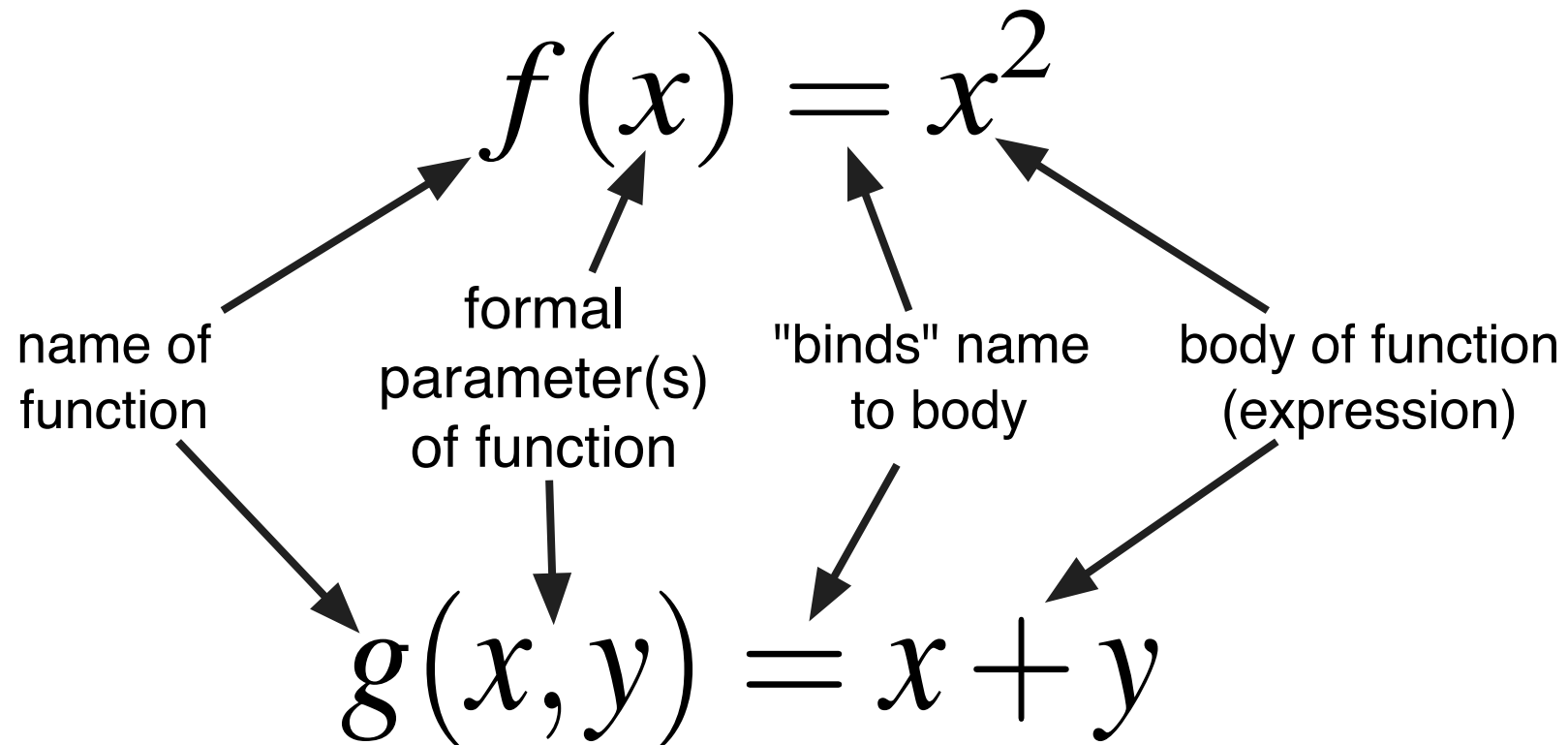
We will not use inexact numbers much (if at all).

Racket expressions causing errors

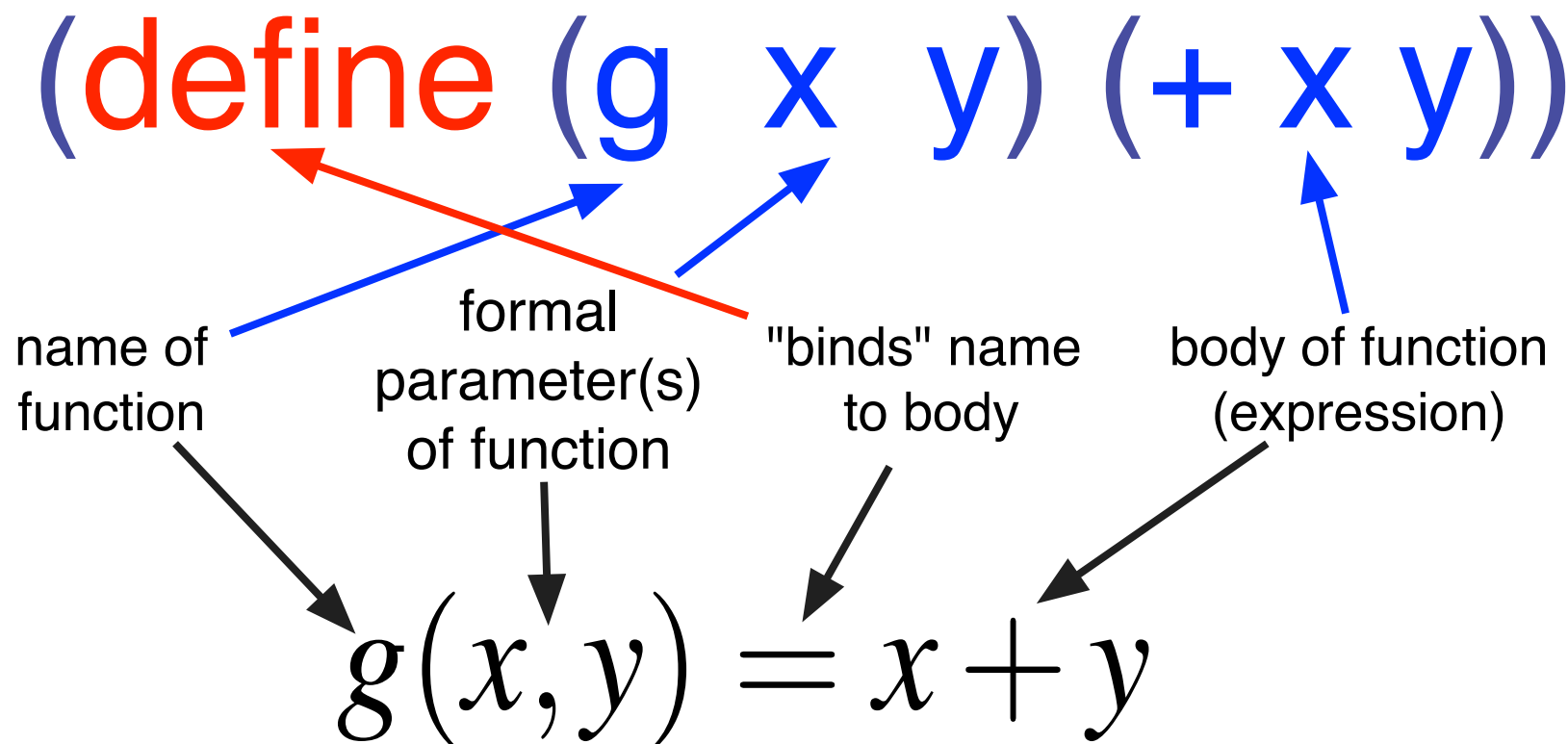
What is wrong with each of the following?

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

Defining functions in mathematics



Defining functions in Racket



Defining functions in Racket

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

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

```
(define (g x y) (+ x y))
```

define is a **special form** (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,
- a single “body” expression.

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

An application of a user-defined function substitutes some values for the corresponding parameters in the definition's expression.

```
(define (g x y) (+ x y))
```

The substitution for `(g 3 5)` would be `(+ 3 5)`.

Evaluating a Racket expression

We use a process of substitution.

Each step is indicated using the ‘yields’ symbol \Rightarrow .

$$(* (- 6 4) (+ 3 2)) \Rightarrow (* 2 (+ 3 2)) \Rightarrow (* 2 5) \Rightarrow 10$$

This mirrors how we work with mathematical expressions.

The same process works with user-defined functions, using the substitution into the body of the function we described earlier.

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

$\Rightarrow (g\ (+\ 1\ 3)\ (f\ 2))$

$\Rightarrow (g\ 4\ (f\ 2))$

$\Rightarrow (g\ 4\ (*\ 2\ 2))$

$\Rightarrow (g\ 4\ 4)$

$\Rightarrow (+\ 4\ 4)$

$\Rightarrow 8$

Each parameter name has meaning only within the body of its function.

```
(define (f x y) (+ x y))
```

```
(define (g x z) (* x z))
```

The two uses of `x` are independent.

Additionally, the following two function definitions define the **same** function:

```
(define (f x y) (+ x y))
```

```
(define (f a b) (+ a b))
```

Defining constants in Racket

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

```
(define k 3)
```

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

The effect of `(define k 3)` is to bind the name `k` to the value 3.

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

In the body of a function, a parameter name “shadows” a constant of the same name.

```
(define x 3)
```

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

$$(+ x x) \Rightarrow (+ 3 x) \Rightarrow (+ 3 3) \Rightarrow 6$$
$$(f 7 6) \Rightarrow (- 7 6) \Rightarrow 1$$
$$(f 5 x) \Rightarrow (f 5 3) \Rightarrow (- 5 3) \Rightarrow 2$$

Advantages of constants

- Can give meaningful names to useful values (e.g. `interest-rate`, `passing-grade`, and `starting-salary`).
- Reduces typing and errors when such values need to be changed
- Makes programs easier to understand
- Constants can be used in any expression, including the body of function definitions
- Sometimes called variables, but their values cannot be changed (until CS 136)

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 let one evaluate expressions step-by-step
- Features: error highlighting, subexpression highlighting, syntax checking

Programs in Racket

A Racket program is a sequence of definitions and expressions.

The expressions are evaluated, using substitution, to produce values.

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

Goals of this module

You should understand the basic syntax of Racket, how to form expressions properly, and what DrRacket might do when given an expression causing an error.

You should be comfortable with these terms: function, parameter, application, argument, constant, expression.

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

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

The design recipe

Readings:

- HtDP, sections 1-5

(ordering of topics is different in lectures, different examples will be used)

- Survival and Style Guides

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

Human-only comments in Racket programs:
from a semicolon (;) to the end of the line.

Some goals for software design

Programs should be:

compatible, composable, correct, durable, efficient, extensible, flexible, maintainable, portable, readable, reliable, reusable, scalable, usable, and useful.

The design recipe

- Use it for every function you write in CS 135.
- A development process that leaves behind written explanation of the development
- Results in a trusted (tested) function which future readers (you or others) can understand

The five design recipe components

Purpose: Describes what the function is to compute.

Contract: Describes what type of arguments the function consumes and what type of value it produces.

Examples: Illustrating the typical use of the function.

Definition: The Racket definition (header and body) of the function.

Tests: A representative set of function arguments and expected function values.

Order of Execution

The order in which you carry out the steps of the design recipe is very important. Use the following order:

- Write a draft of the Purpose
- Write Examples (by hand, then code)
- Write Definition Header & Contract
- Finalize the purpose with parameter names
- Write Definition Body
- Write Tests

Using the design recipe

We'll write a function which squares two numbers and sums the results.

Purpose:

:: (sum-of-squares p1 p2) produces the sum of
:: the squares of p1 and p2

Contract:

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

Mathematically: sum-of-squares: $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$

:: Examples:

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

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

Header & Body:

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

:: Tests:

```
(check-expect (sum-of-squares 0 0) 0)
```

```
(check-expect (sum-of-squares -2 7) 53)
```

Contracts

- We will be more careful than HtDP and use abbreviations.
- **Num**: any Racket numeric value
- **Int**: restriction to integers
- **Nat**: restriction to natural numbers (including 0)
- **Any**: any Racket value
- We will see more types soon.

Tests

- Tests should be written later than the code body.
- Tests can then handle complexities encountered while writing the body.
- Tests don't need to be “big”.
In fact, they should be small and directed.
- The number of tests and examples needed is a matter of judgement.
- **Do not** figure out the expected answers to your tests by running your program! Always work them out **by hand**.

The teaching languages offer a convenient testing method.

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

```
(check-within (sqrt 2) 1.414 .001)
```

```
(check-error (/ 1 0) "?: division by zero")
```

Tests written using these functions are saved and evaluated at the very end of your program.

This means that examples can be written as code.

:: (sum-of-squares p1 p2) produces the sum of

:: the squares of p1 and p2

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

(check-expect (sum-of-squares 0 0) 0)

(check-expect (sum-of-squares -2 7) 53)

Additional contract requirements

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

```
:: (my-function a b c) ...
```

```
:: my-function: Num Num Num  $\rightarrow$  Num
```

```
:: requires:  $0 < a < b$ 
```

```
::           c must be non-zero
```

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 before completing your assignments.

Boolean-valued functions

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

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

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

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

Standard Racket uses `#t` and `#f`; these will sometimes show up in basic tests and correctness tests.

Other types of comparisons

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

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

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

$:: = : \text{Num Num} \rightarrow \text{Bool}$

Note that Boolean is abbreviated in contracts.

Complex relationships

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

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

These are used to test complex relationships.

Example: the proposition “ $3 \leq x < 7$ ”, which is the same as “ $x \in [3, 7)$ ”, can be computationally tested by evaluating **(and (<= 3 x) (< x 7))**.

Some computational differences

The mathematical AND, OR connect two propositions.

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

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

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

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

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

Examples:

```
(and (not (= x 0)) (<= (/ y x) c))
```

```
(or (= x 0) (> (/ y x) c))
```

These will never divide by zero.

Predicates

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

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

We can write our own:

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

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

Symbolic data

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

A symbol is defined using an apostrophe or 'quote': 'blue

The symbol 'blue is a value just like 6, but it is more limited computationally.

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

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

```
(define my-symbol 'blue)
```

```
(symbol=? my-symbol 'red)  $\Rightarrow$  false
```

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

Other types of data

Racket also supports strings, such as "blue".

What are the differences between strings and symbols?

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

Here are a few functions which operate on strings.

`(string-append "alpha" "bet")` \Rightarrow `"alphabet"`

`(string-length "perpetual")` \Rightarrow `9`

`(string<? "alpha" "bet")` \Rightarrow `true`

The textbook does not use strings; it uses symbols.

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

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

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

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

General equality testing

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

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.

Do not overuse `equal?`, however.

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

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

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

Conditional expressions

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

Example: taking the absolute value of x .

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

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

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 function in Racket

The general form of a conditional expression is

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

where `questionk` could be `else`.

The questions are evaluated in 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.
(the one associated with the first question that evaluates to **true**, or associated with the **else** if that is present and reached)

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

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

Simplifying conditional functions

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

Suppose our analysis identifies three intervals:

- students who fail CS 135 must take CS 115 (this isn't true).
- students who pass but get less than 60% go into CS 116.
- students who pass and get at least 60% go into CS 136.

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

```
(define (course-after-cs135 grade)
  (cond [(< grade 50) 'cs115]
        [(and (>= grade 50) (< grade 60)) 'cs116]
        [(>= grade 60) 'cs136]))
```

We can simplify the second and third tests.

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

These simplifications become second nature with practice.

Tests for conditional expressions

- Write at least one test for each possible answer in the expression.
- That test should be simple and direct, aimed at testing that answer.
- Often tests are appropriate at boundary points as well.
- DrRacket highlights unused code.

For the example above:

```
(cond [(< grade 50) 'cs115]  
      [(< grade 60) 'cs116]  
      [else 'cs136]))
```

there are three intervals and two boundary points, so five tests are required (for instance, 40, 50, 55, 60, 70).

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

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

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

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

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

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

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

Writing Boolean tests

The textbook writes tests in this fashion:

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

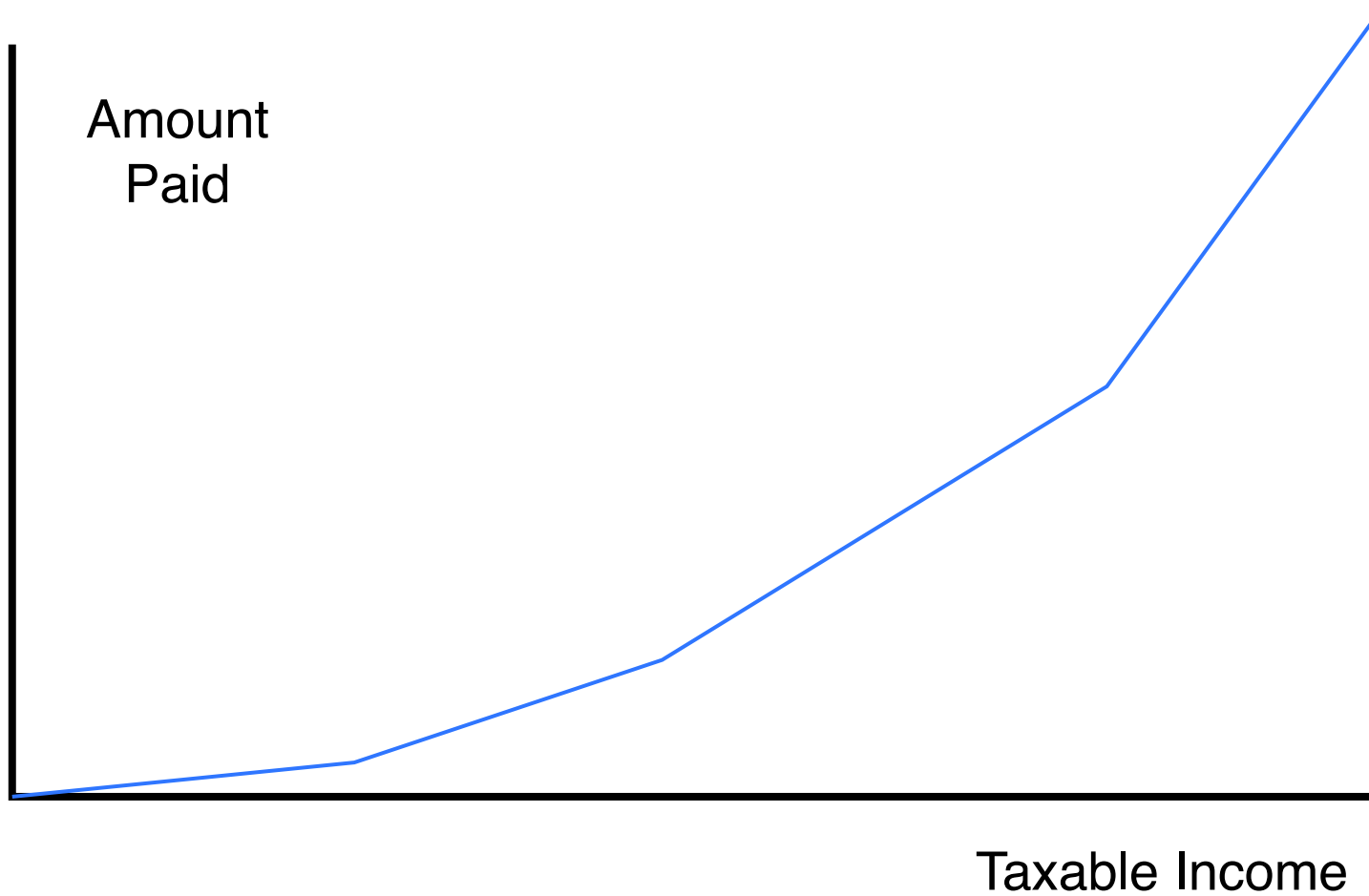
which works outside the teaching languages.

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

Example: computing taxes

Canada has a **progressive** tax system: the rate of tax increases with income. For 2014, the rates are:

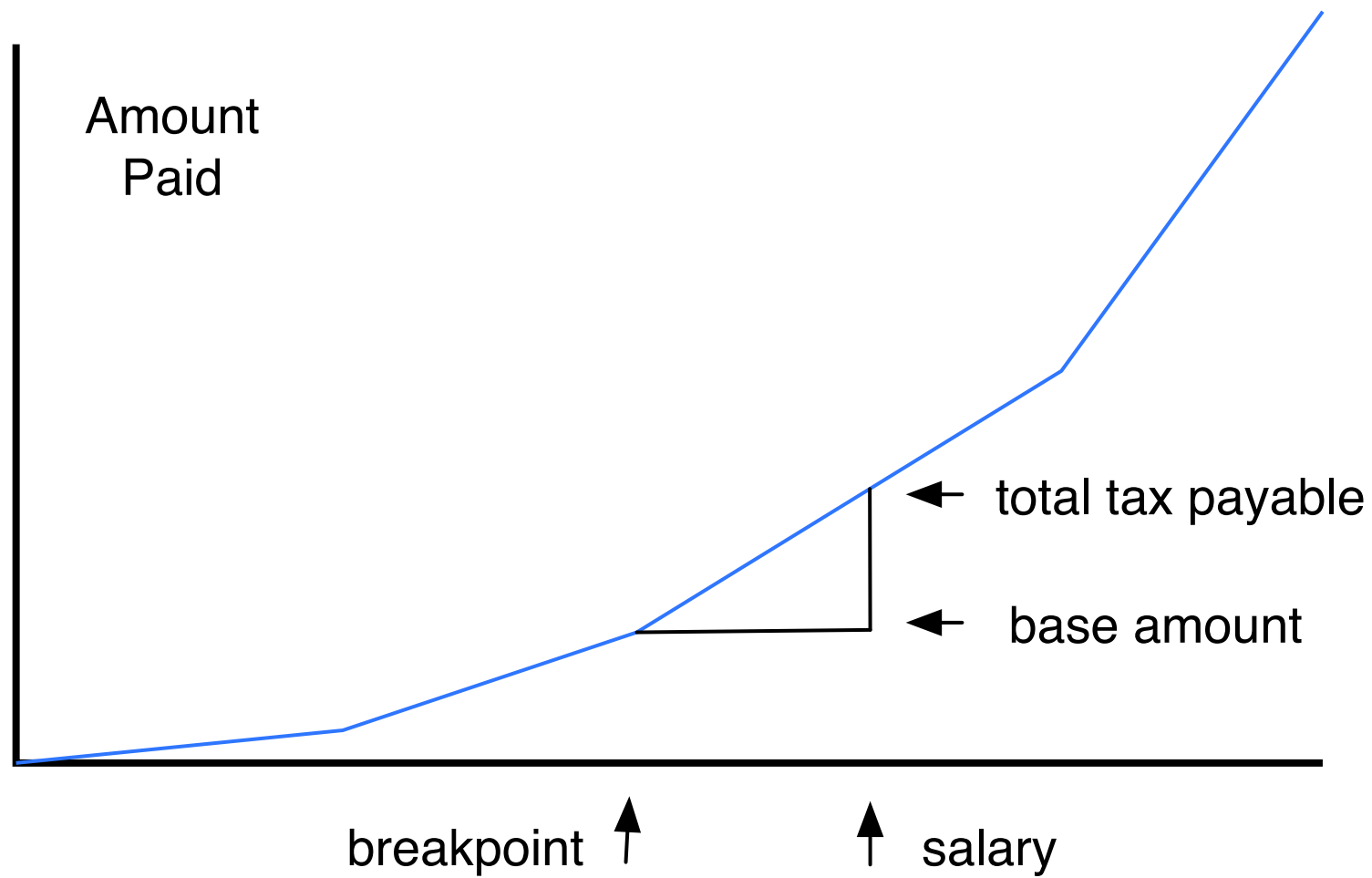
- no tax payable on negative income
- 15% on the amount from \$0 to \$43,953
- 22% on the amount from \$43,953 to \$87,907
- 26% on the amount from \$87,907 to \$136,270
- 29% on the amount from \$136,270 and up



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

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

This is what the paper Canadian tax form does.



:: breakpoints

(define bp1 43953)

(define bp2 87907)

(define bp3 136270)

:: rates

(define rate1 0.15)

(define rate2 0.22)

(define rate3 0.26)

(define rate4 0.29)

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

;; base_i is the base amount for interval [bp_i,bp_(i+1)]

;; that is, tax payable at income bp_i

```
(define base1 (* bp1 rate1))
```

```
(define base2 (+ base1 (* (- bp2 bp1) rate2)))
```

```
(define base3 (+ base2 (* (- bp3 bp2) rate3)))
```

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


Helper functions

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

```
(define (tax-calc base low high rate)  
  (+ base (* (- high low) rate)))
```

This can be used both in the definition of constants and in the main function.

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

Good example: movie theatre (section 3.1 of HtDP).

Helper functions generalize similar expressions.

They avoid long, unreadable function definitions.

Use judgement: don't go overboard, but sometimes very short definitions improve readability.

Helper functions must also follow the design recipe

Give all functions (including helpers) meaningful names, not “helper”

Goals of this module

You should understand the reasons for each of the components of the design recipe and the particular way that they are expressed.

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

You should understand Boolean data, and be able to perform and combine comparisons to test complex conditions on numbers.

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

You should understand how to write `check-expect` examples and tests, and use them in your assignment submissions.

You should be aware of other types of data (symbols and strings), which will be used in future lectures.

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

The syntax and semantics of Beginning Student

Readings: HtDP, Intermezzo 1 (Section 8).

We are covering the ideas of section 8, but not the parts of it dealing with section 6/7 material (which will come later), and in a somewhat different fashion.

- A program has a precise meaning and effect.
- A model of a programming language provides a way of describing the meaning of a program.
- Typically this is done informally, by examples.
- With Racket, we can do better.

Advantages in modelling Racket

- Few language constructs, so model description is short
- We don't need anything more than the language itself!
 - No diagrams
 - No vague descriptions of the underlying machine

Spelling rules for Beginning Student

Identifiers are the names of constants, parameters, and user-defined functions.

They are made up of letters, numbers, hyphens, underscores, and a few other punctuation marks. They must contain at least one non-number. They can't contain spaces or any of these:

() , ; { } [] ' ' " " .

Symbols start with a single quote ' followed by something obeying the rules for identifiers.

There are rules for numbers (integers, rationals, decimals) which are fairly intuitive.

There are some built-in constants, like `true` and `false`.

Of more interest to us are the rules describing program structure.

For example: a program is a sequence of definitions and expressions.

Syntax and grammar

There are three problems we need to address:

1. Syntax: The way we're allowed to say things.
'?is This Sentence Syntactically Correct'
2. Semantics: the meaning of what we say.
'Trombones fly hungrily.'
3. Ambiguity: valid sentences have exactly one meaning.
'Sally was given a book by Joyce.'

English rules on these issues are pretty lax. For Racket, we need rules that *always* avoid these problems.

Grammars

To enforce syntax and avoid ambiguity, we'd *like* to use grammars.

For example, an English sentence can be made up of a subject, verb, and object, in that order.

We might express this as follows:

$$\langle \text{sentence} \rangle = \langle \text{subject} \rangle \langle \text{verb} \rangle \langle \text{object} \rangle$$

The linguist Noam Chomsky formalized grammars in this fashion in the 1950's. The idea proved useful for programming languages.

The textbook describes function definitions like this:

$$\langle \text{def} \rangle = (\text{define } (\langle \text{var} \rangle \langle \text{var} \rangle \dots \langle \text{var} \rangle) \langle \text{exp} \rangle)$$

There is a similar rule for defining constants. Additional rules define **cond** expressions, etc.

When we've tried this in the past students have found grammars difficult. Furthermore, the documentation materials use two different approaches. The Help Desk presents the same idea as

`definition = (define (id id id ...) expr)`

So, we will use informal descriptions instead.

CS 241, CS 230, CS 360, and CS 444 discuss the mathematical formalization of grammars and their role in the interpretation of computer programs and other structured texts.

Semantic Model

The second of our three problems (syntax, semantics, ambiguity) we will solve rigorously with a semantic model. A semantic model of a programming language provides a method of predicting the result of running any program.

Our model will repeatedly simplify the program via substitution. Every substitution step yields a valid Racket program, until all that remains is a sequence of definitions and values.

A substitution step finds the leftmost subexpression eligible for rewriting, and rewrites it by the rules we are about to describe.

Using an ellipsis

To express our substitution rules, we'll need to use an ellipsis.

An ellipsis (. . .) is used in English to indicate an **omission**.

“The Prime Minister said that the Opposition was. . . right.”

In mathematics, an ellipsis is often used to indicate a **pattern**.

“The positive integers less than n are $1, 2, \dots, n - 1$.”

We will use both types of ellipsis in this course, as well as some new ones.

Application of built-in functions

We reuse the rules for the arithmetic expressions we are familiar with to substitute the appropriate value for expressions like $(+ 3 5)$ and $(\text{expt } 2 10)$.

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

$$(\text{expt } 2 10) \Rightarrow 1024$$

Formally, the substitution rule is:

$(f\ v_1 \dots v_n) \Rightarrow v$ where f is a built-in function and v the value of $f(v_1, \dots, v_n)$.

Note the two uses of a pattern ellipsis.

Application of user-defined functions

As an example, consider `(define (term x y) (* x (sqr y)))`.

The function application `(term 2 3)` can be evaluated by taking the body of the function definition and replacing `x` by 2 and `y` by 3.

The result is `(* 2 (sqr 3))`.

The rule does not apply if an argument is not a value, as in the case of the second argument in `(term 2 (+ 1 2))`.

Any argument which is not a value must first be simplified to a value using the rules for expressions.

The general substitution rule is:

$$(f\ v1\ \dots\ vn) \Rightarrow exp'$$

where `(define (f x1 ... xn) exp)` occurs to the left, and `exp'` is obtained by substituting into the expression `exp`, with all occurrences of the formal parameter `xi` replaced by the value `vi` (for `i` from 1 to `n`).

Note we are using a pattern ellipsis in the rules for both built-in and user-defined functions to indicate several arguments.

An example

```
(define (term x y) (* x (sqr y)))
```

```
(term (− 3 1) (+ 1 2))
```

```
⇒ (term 2 (+ 1 2))
```

```
⇒ (term 2 3)
```

```
⇒ (* 2 (sqr 3))
```

```
⇒ (* 2 9)
```

```
⇒ 18
```

A constant definition binds a name (the constant) to a value (the value of the expression).

We add the substitution rule:

$id \Rightarrow val$, where $(\text{define } id \text{ } val)$ occurs to the left.

An example

`(define x 3) (define y (+ x 1)) y`

\Rightarrow

`(define x 3) (define y (+ 3 1)) y`

\Rightarrow

`(define x 3) (define y 4) y`

\Rightarrow

`(define x 3) (define y 4) 4`

Substitution in cond expressions

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

$$(\text{cond} [\text{false exp}] \dots) \Rightarrow (\text{cond} \dots).$$
$$(\text{cond} [\text{true exp}] \dots) \Rightarrow \text{exp}.$$
$$(\text{cond} [\text{else exp}]) \Rightarrow \text{exp}.$$

These suffice to simplify any **cond** expression.

Here we are using an omission ellipsis to avoid specifying the remaining clauses in the **cond**.

An example

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

Error: `y` undefined

Errors

A syntax error occurs when a sentence cannot be interpreted using the grammar.

A run-time error occurs when an expression cannot be reduced to a value by application of our (still incomplete) evaluation rules.

Example: (`cond` [`(> 3 4)` `x`])

Simplification Rules for **and** and **or**

The simplification rules we use for Boolean expressions involving **and** and **or** are different from the ones the Stepper in DrRacket uses.

The end result is the same, but the intermediate steps are different.

The implementers of the Stepper made choices which result in more complicated rules, but whose intermediate steps appear to help students in lab situations.

$(\text{and false } \dots) \Rightarrow \text{false}.$

$(\text{and true } \dots) \Rightarrow (\text{and } \dots).$

$(\text{and}) \Rightarrow \text{true}.$

$(\text{or true } \dots) \Rightarrow \text{true}.$

$(\text{or false } \dots) \Rightarrow (\text{or } \dots).$

$(\text{or}) \Rightarrow \text{false}.$

As in the rewriting rules for **cond**, we are using an omission ellipsis.

Substitution Rules (so far)

1. Apply functions only when all arguments are values.
2. When given a choice, evaluate the leftmost expression first.
3. $(f\ v1\dots vn) \Rightarrow v$ when f is built-in...
4. $(f\ v1\dots vn) \Rightarrow \text{exp}'$ when $(\text{define } (f\ x1\dots xn)\ \text{exp})$ occurs to the left...
5. $\text{id} \Rightarrow \text{val}$ when (define id val) occurs to the left.

6. $(\text{cond} [\text{false exp}] \dots) \Rightarrow (\text{cond} \dots)$.
7. $(\text{cond} [\text{true exp}] \dots) \Rightarrow \text{exp}$.
8. $(\text{cond} [\text{else exp}]) \Rightarrow \text{exp}$.
9. $(\text{and false} \dots) \Rightarrow \text{false}$.
10. $(\text{and true} \dots) \Rightarrow (\text{and} \dots)$.
11. $(\text{and}) \Rightarrow \text{true}$.
12. $(\text{or true} \dots) \Rightarrow \text{true}$.
13. $(\text{or false} \dots) \Rightarrow (\text{or} \dots)$.
14. $(\text{or}) \Rightarrow \text{false}$.

Importance of the model

We will add to the semantic model when we introduce a new feature of Racket.

Understanding the semantic model is very important in understanding the meaning of a Racket program.

Doing a step-by-step reduction according to these rules is called **tracing** a program.

It is an important skill in any programming language or computational system.

We will test this skill on assignments and exams.

Unfortunately, once we start dealing with unbounded data, traces get very long, and the intermediate steps are hard to make sense of in the Stepper.

In future traces, we will often do a **condensed trace** by skipping several steps in order to put the important transformations onto one slide.

It is very important to know when these gaps occur and to be able to fill in the missing transformations.

Goals of this module

You should understand the substitution-based semantic model of Racket, and be prepared for future extensions.

You should be able to trace the series of simplifying transformations of a Racket program.

Structures

Readings: HtDP, sections 6, 7.

- Avoid 6.2, 6.6, 6.7, 7.4.
- These use the obsolete `draw.ss` teachpack.
- The new `image.ss` and `world.ss` are more functional.

Compound data

The teaching languages provide a general mechanism called **structures**.

They permit the “bundling” of several values into one.

In many situations, data is naturally grouped, and most programming languages provide some mechanism to do this.

There is also one predefined structure, `posn`, to provide an example.

Posn structures

- **constructor** function `make-posn`, with contract
;; `make-posn`: Num Num \rightarrow Posn
- **selector** functions `posn-x` and `posn-y`, with contracts
;; `posn-x`: Posn \rightarrow Num
;; `posn-y`: Posn \rightarrow Num

Example:

```
(define mypoint (make-posn 8 1))
```

```
(posn-x mypoint)  $\Rightarrow$  8
```

```
(posn-y mypoint)  $\Rightarrow$  1
```

Possible uses:

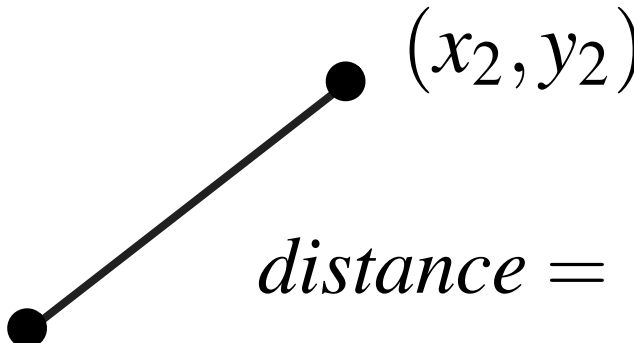
- coordinates of a point on a two-dimensional plane
- positions on a screen or in a window
- a geographical position

An expression such as `(make-posn 8 1)` is considered a value.

This expression will not be rewritten by the Stepper or our semantic rules.

The expression `(make-posn (+ 4 4) (- 3 2))` would be rewritten to (eventually) yield `(make-posn 8 1)`.

Example: point-to-point distance


$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

:: (distance posn1 posn2) computes the Euclidean distance

:: between posn1 and posn2

:: distance: Posn Posn \rightarrow Num

:: Example:

```
(check-expect (distance (make-posn 1 1) (make-posn 4 5))  
               5)
```

```
(define (distance posn1 posn2)  
  (sqrt (+ (sqr (− (posn-x posn2) (posn-x posn1)))  
           (sqr (− (posn-y posn2) (posn-y posn1))))))
```

Functions which produce posns

:: (scale point factor) scales point by the given factor

:: scale: Posn Num \rightarrow Posn

:: Example:

```
(check-expect (scale (make-posn 3 4) 0.5)
               (make-posn 1.5 2))
```

```
(define (scale point factor)
  (make-posn (* factor (posn-x point))
              (* factor (posn-y point))))
```


Misusing posns

What is the result of evaluating the following expression?

```
(distance (make-posn 'Iron 'Man)  
          (make-posn 'Tony 'Stark))
```

This causes a run-time error, but at a surprising point.

Racket does not enforce contracts, which are just comments, and ignored by the machine.

Each value created during the running of a program has a type (integer, Boolean, etc.).

Types are associated with values, not with constants or parameters.

```
(define p 5)
```

```
(define q (mystery-fn 5))
```

This is known as **dynamic typing**.

Many other mainstream languages use a more restrictive approach known as **static typing**.

With static typing, the header of our `distance` function might look like this:

```
real distance(Posn posn1, Posn posn2)
```

Here the contract is part of the language.

A program containing the function application

`distance(3, "test")` would be illegal.

Dynamic typing is a potential source of both flexibility (as we will see) and confusion.

To avoid making mistakes such as the one with `make-posn`, we can use **data definitions**:

`:: A Posn is a (make-posn Num Num)`

Definitions like this are human-readable comments and not enforced by the machine.

We can also create functions which check their arguments to catch type errors more gracefully (examples soon).

Defining structures

If `posn` wasn't built in, we could define it:

```
(define-struct posn (x y))
```

The arguments to the `define-struct` special form are:

- a structure name (e.g. `posn`), and
- a list of field names in parentheses.

Doing this once creates a number of functions that can be used many times.

The expression `(define-struct posn (x y))` creates:

- **Constructor:** `make-posn`
- **Selectors:** `posn-x`, `posn-y`
- **Predicate:** `posn?`

The `posn?` predicate tests if its argument is a `posn`.

Structures were added to the teaching languages before they were added to standard Racket in 2007.

It is not hard to build structures using other Racket features.

We will see a few ways to do it in CS 135 and CS 136.

Later on, we will see how structures can be viewed as the basis for objects, the main way of handling data and of organizing larger programs in many languages.

Stepping with structures

The special form

```
(define-struct sname (fname1 ... fnamen))
```

defines the structure type **sname** and automatically defines the following primitive functions:

- **Constructor:** **make-sname**
- **Selectors:** **sname-fname1 ... sname-fnamen**
- **Predicate:** **sname?**

Sname may be used in contracts.

The substitution rule for the i th selector is:

$(\text{pname-fname}_i (\text{make-pname } v_1 \dots v_i \dots v_n)) \Rightarrow v_i.$

Finally, the substitution rules for the new predicate are:

$(\text{pname? } (\text{make-pname } v_1 \dots v_n)) \Rightarrow \text{true}$

$(\text{pname? } V) \Rightarrow \text{false}$ for V a value of any other type.

In these rules, we use a pattern ellipsis.

An example using posns

```
(define myposn (make-posn 4 2))
```

```
(scale myposn 0.5) ⇒
```

```
(scale (make-posn 4 2) 0.5) ⇒
```

```
(make-posn
```

```
  (* 0.5 (posn-x (make-posn 4 2)))
```

```
  (* 0.5 (posn-y (make-posn 4 2)))) ⇒
```

```
(make-posn
```

```
  (* 0.5 4)
```

```
  (* 0.5 (posn-y (make-posn 4 2)))) ⇒
```

(make-posn 2 (* 0.5 (posn-y (make-posn 4 2)))) \Rightarrow

(make-posn 2 (* 0.5 2)) \Rightarrow

(make-posn 2 1)

Data definition and analysis

Suppose we want to represent information associated with downloaded MP3 files.

- The name of the performer
- The title of the song
- The length of the song
- The genre of the music (rap, country, etc.)

The data definition on the next slide will give a name to each field and associate a type of data with it.

Structure and Data Defs for Mp3Info

```
(define-struct mp3info (performer title length genre))
```

```
:: An Mp3Info is a (make-mp3info Str Str Num Sym)
```

This creates the following functions:

- constructor `make-mp3info`,
- selectors `mp3info-performer`, `mp3info-title`, `mp3info-length`, `mp3info-genre`, and
- type predicate `mp3info?`.

Templates and data-directed design

One of the main ideas of the HtDP textbook is that the form of a program often mirrors the form of the data.

A template is a general framework within which we fill in specifics.

We create a template once for each new form of data, and then apply it many times in writing functions that consume that type of data.

A template is derived from a data definition.

Templates for compound data

The template for a function that consumes a structure selects every field in the structure, though a specific function may not use all the selectors.

```
:: my-mp3info-fn: Mp3Info → Any  
(define (my-mp3info-fn info)  
  (... (mp3info-performer info) ...  
    (mp3info-title info) ...  
    (mp3info-length info) ...  
    (mp3info-genre info) ... ))
```

An example

:: (correct-performer oldinfo newname) corrects the name

:: in oldinfo to be newname

:: correct-performer: Mp3Info Str → Mp3Info

:: Example:

(check-expect

 (correct-performer

 (make-mp3info "Hannah Montana" "Bite This" 80 'Punk)
 "Anonymous Doner Kebab")

 (make-mp3info "Anonymous Doner Kebab" "Bite This"
 80 'Punk))

:: correct-performer: Mp3Info Str → Mp3Info

```
(define (correct-performer oldinfo newname)
  (make-mp3info
    newname
    (mp3info-title oldinfo)
    (mp3info-length oldinfo)
    (mp3info-genre oldinfo)))
```

We could easily have done this without a template, but the use of a template pays off when designing more complicated functions.

Stepping the example

```
(define mymp3 (make-mp3info "Avril Lavigne" "Boy" 180 'Rock))  
(correct-performer mymp3 "U2") ⇒  
(correct-performer  
  (make-mp3info "Avril Lavigne" "Boy" 180 'Rock) "U2") ⇒  
(make-mp3info  
  "U2"  
  (mp3info-title (make-mp3info "Avril Lavigne" "Boy" 180 'Rock))  
  (mp3info-length (make-mp3info "Avril Lavigne" "Boy" 180 'Rock))  
  (mp3info-genre (make-mp3info "Avril Lavigne" "Boy" 180 'Rock)))  
⇒ (make-mp3info "U2" "Boy" 180 'Rock); after three steps
```

Design recipe for compound data

Do this *once per new structure type*:

Data Analysis and Definition: Define any new structures needed, based on problem description. Write data definitions for the new structures.

Template: Created once for each structure type, used for functions that consume that type.

Design recipe for compound data

Do the usual design recipe *for every function*:

Purpose: Same as before.

Contract: Can use both atomic data types and defined structure names.

Examples: Same as before.

Definition: To write the body, expand the template based on examples.

Tests: Same as before. Be sure to capture all cases.

Dealing with mixed data

Racket provides predicates to identify data types, such as `number?` and `symbol?`

`define-struct` also creates a predicate that tests whether its argument is that type of structure (e.g. `posn?`).

We can use these to check aspects of contracts, and to deal with data of mixed type.

Example: multimedia files

```
(define-struct movieinfo (director title duration genre))  
;; A MovieInfo is a (make-movieinfo Str Str Num Sym )  
;;  
;; A MmInfo is one of:  
;; ★ an Mp3Info  
;; ★ a MovieInfo
```

Here “mm” is an abbreviation for “multimedia”.

The template for mminfo

The template for mixed data is a **cond** with each type of data, and if the data is a structure, we apply the template for structures.

```
:: my-mminfo-fn: Mminfo → Any
```

```
(define (my-mminfo-fn info)
```

```
  (cond [(mp3info? info)
```

```
    (... (mp3info-performer info) ...
```

```
      (mp3info-title info) ... ]); two more fields
```

```
[(movieinfo? info)
```

```
  (... (movieinfo-director info) ... ]))]; three more fields
```

```
(define favsong (make-mp3info "Beck" "Tropicalia"  
                             185 'Alternative))
```

```
(define favmovie (make-movieinfo "Orson Welles" "Citizen Kane"  
                                 119 'Classic))
```

:: (mminfo-artist info) produces performer/director name from info

:: mminfo-artist: MmInfo \rightarrow Str

:: Examples:

```
(check-expect (mminfo-artist favsong) "Beck")
```

```
(check-expect (mminfo-artist favmovie) "Orson Welles")
```



```
(define (mminfo-artist info)
  (cond [(mp3info? info) (mp3info-performer info)]
        [(movieinfo? info)(movieinfo-director info)]))
```

The point of the design recipe and the template design:

- to make sure that one understands the type of data being consumed and produced by the function
- to take advantage of common patterns in code

anyof types

Unlike Mp3Info and MovieInfo, there is no **define-struct** expression associated with **MmInfo**.

For the contract

```
:: mminfo-artist: MmInfo → Str
```

to make sense, the data definition for MmInfo must be included as a comment in the program.

Another option is to use the notation

```
:: mminfo-artist: (anyof Mp3Info MovieInfo) → Str
```

Checked functions

We can write a safe version of `make-posn`.

`:: safe-make-posn: Num Num \rightarrow Posn`

```
(define (safe-make-posn x y)
  (cond [(and (number? x) (number? y)) (make-posn x y)]
        [else (error "numerical arguments required")]))
```

The application `(safe-make-posn 'Tony 'Stark)` produces the error message “numerical arguments required”.

We were able to form the `MmInfo` type because of Racket's dynamic typing.

Statically-typed languages need to offer some alternative method of dealing with mixed data.

In later CS courses, you will see how the object-oriented features of inheritance and polymorphism gain some of this flexibility, and handle some of the checking we have seen in a more automatic fashion.

Goals of this module

You should understand the use of `posns`.

You should be able to write code to define a structure, and to use the functions that are defined when you do so.

You should understand the data definitions we have used, and be able to write your own.

You should be able to write the template associated with a structure definition, and to expand it into the body of a particular function that consumes that type of structure.

You should understand the use of type predicates and be able to write code that handles mixed data.

You should understand (`anyof . . .`) notation in contracts. We will use it in later lecture modules.

Lists

Readings: HtDP, sections 9 and 10.

- Avoid 10.3 (uses `draw.ss`).

Introducing lists

Structures are useful for representing a fixed amount of data.

But there are many circumstances in which the amount of data is unbounded, meaning it may grow or shrink – and you don't know how much.

For example, suppose you enjoy attending concerts of local musicians and want a list of the upcoming concerts you plan to attend. The number will change as time passes.

We will also be concerned about order: which concert is the first one to attend, the next, and so on.

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

- A list of 4 concerts is a concert followed by a list of 3 concerts.
- A list of 3 concerts is a concert followed by a list of 2 concerts.
- A list of 2 concerts is a concert followed by a list of 1 concert.
- A list of 1 concert is a concert followed by a list of 0 concerts.

A list of zero concerts is special. We'll call it the empty list.

Basic list constructs

- **empty**: A value representing a list with 0 items.
- **cons**: Consumes an item and a list and produces a new, longer list.
- **first**: Consumes a nonempty list and produces the first item.
- **rest**: Consumes a nonempty list and produces the same list without the first item.
- **empty?**: Consumes a value and produces **true** if it is **empty** and **false** otherwise.
- **cons?**: Consumes a value and produces **true** if it is a **cons** value and **false** otherwise.

List data definition

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

:: A list is one of:

:: ★ **empty**

:: ★ **(cons value list)**

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

Lists are the main data structure in standard Racket.

Example lists

`(define clst empty)` is a sad state of affairs – no upcoming concerts.

`(define clst1 (cons 'Waterboys empty))` is a list with one concert to attend.

`(define clst2 (cons 'DaCapo clst1))` is a new list just like `clst1` but with a new concert at the beginning.

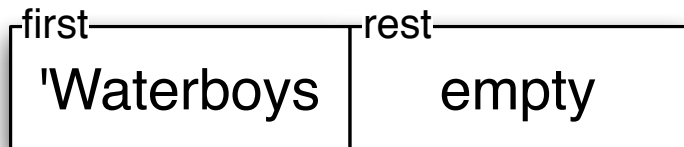
`(define clst2alt (cons 'DaCapo (cons 'Waterboys empty)))` is just like `clst2`.

`(define clst3 (cons 'Waterboys (cons 'DaCapo (cons 'Waterboys empty))))` is a list with two Waterboys and one DaCapo concert.

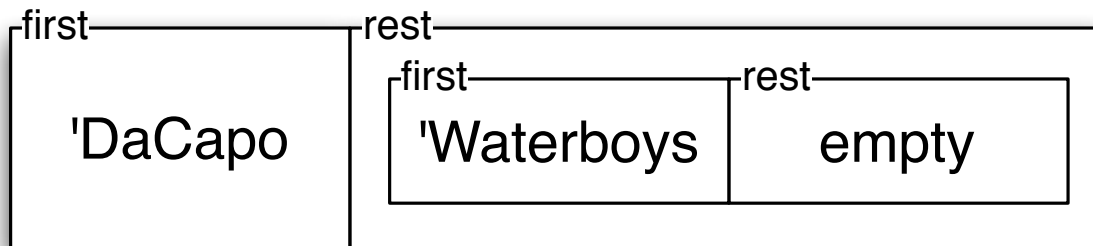
Nested boxes visualization

`cons` can be thought of as producing a two-field structure. It can be visualized two ways. The first:

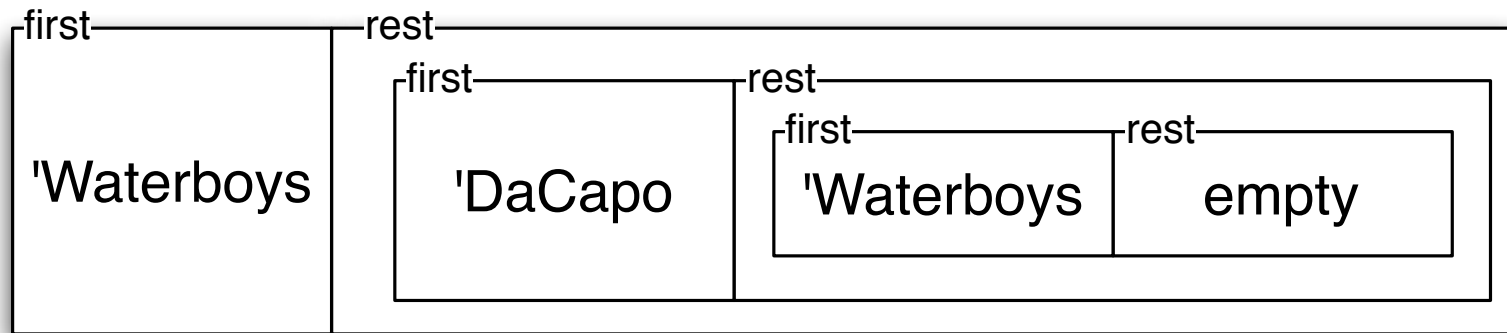
```
(cons 'Waterboys empty)
```



```
(cons 'DaCapo (cons 'Waterboys empty))
```



```
(cons 'Waterboys  
  (cons 'DaCapo  
    (cons 'Waterboys  
      empty))))
```

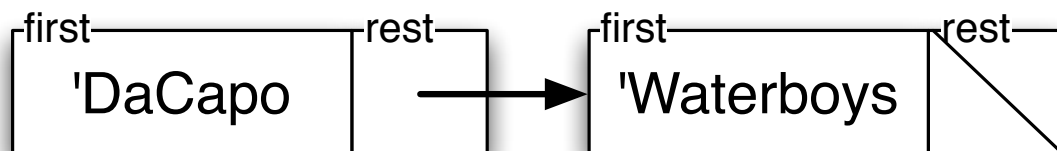


Box-and-pointer visualization

(cons 'Waterboys empty)



(cons 'DaCapo (cons 'Waterboys empty))



(cons 'Waterboys (cons 'DaCapo (cons 'Waterboys empty)))



Extracting values from a list

```
(define clst (cons 'Waterboys  
                  (cons 'DaCapo (cons 'Waterboys empty))))
```

First concert:

```
(first clst) ⇒ 'Waterboys
```

Concerts after the first:

```
(rest clst) ⇒ (cons 'DaCapo (cons 'Waterboys empty))
```

Second concert:

```
(first (rest clst)) ⇒ 'DaCapo
```


Contracts

The set of types DrRacket knows (and that we have studied so far) includes `Num`, `Int`, `Sym`, `Str`, and `Bool`. These terms can all be used in function contracts.

We also use terms in contracts that DrRacket does not know about. These include `anyof` types, names we have defined in a data definition (like `MP3Info` for a structure), and now lists.

(listof X) notation in contracts

We'll use (listof X) in contracts, where X may be replaced with any type. For the concert list example in the previous slides, the list contains only symbols and has type (listof Sym).

Other examples: (listof Num), (listof Bool), (listof (anyof Num Sym)), (listof MP3Info), and (listof Any).

Replace X with the most specific type available.

Semantics of list functions

`(cons a b)` is a value.

- `a` must be a value
- There are no restrictions on the values of the first argument, allowing us to mix data types in a list (and even have lists of lists).
- `b` must be a list (`empty` is a list)
- Like the values `1`, `'Waterboys`, and `(make-posn 1 5)`, `(cons a b)` will not be simplified.

The substitution rules for `first`, `rest`, and `empty?` are:

- $(\text{first } (\text{cons } a \ b)) \Rightarrow a$, where a and b are values.
- $(\text{rest } (\text{cons } a \ b)) \Rightarrow b$, where a and b are values.
- $(\text{empty? } \text{empty}) \Rightarrow \text{true}$.
- $(\text{empty? } a) \Rightarrow \text{false}$ where a is not `empty`

Functions on lists

Using these built-in functions, we can write our own simple functions on lists.

:: (next-concert los) produces the next concert to attend or

:: false if los is empty

:: next-concert: (listof Sym) \rightarrow (anyof false Sym)

(check-expect (next-concert (cons 'a (cons 'b empty))) 'a)

(check-expect (next-concert empty) false)

```
(define (next-concert los)
  (cond [(empty? los) false]
        [else (first los)]))
```

:: (same-consec? los) determines if next two concerts are the same

:: same-consec?: (listof Sym) \rightarrow Bool

(check-expect (same-consec? (cons 'a (cons 'b empty))) false)

(check-expect (same-consec? (cons 'a (cons 'a empty))) true)

(check-expect (same-consec? (cons 'a empty)) false)

(define (same-consec? los)

(and (not (empty? los))

(not (empty? (rest los)))

(symbol=? (first los) (first (rest los)))))

Processing lists

Most interesting functions will process the entire consumed list. How many concerts are on the list? How many times does 'Waterboys appear? Which artists are duplicated in the list?

We build a template from our data definition.

:: A list of symbols, (listof Sym), is one of:

:: ★ empty

:: ★ (cons Sym (listof Sym))

Template for processing a list of symbols

`:: my-los-fn: (listof Sym) \rightarrow Any`

```
(define (my-los-fn los)
  (cond [(empty? los) ...]
        [(cons? los) ...]))
```

The second test can be replaced by **else**.

Because **cons** is a special type of structure, we can add the selectors as in the structure template.

:: my-los-fn: (listof Sym) \rightarrow Any

```
(define (my-los-fn los)
  (cond [(empty? los) ...]
        [else (... (first los) ... (rest los) ...)]))
```

Now we go a step further.

Since `(rest los)` also is of type `(listof Sym)`, we apply the same computation to it – that is, we apply `my-los-fn`.

Here is the resulting template for a function consuming a `(listof Sym)`, which matches the data definition:

```
:: my-los-fn: (listof Sym) → Any
```

```
(define (my-los-fn los)
  (cond [(empty? los) ...]
        [else (... (first los) ...
                     (my-los-fn (rest los)) ... )]))
```

We can now fill in the dots for a specific example.

Example: how many concerts?

;; (count-concerts los) counts the number of concerts in los

;; count-concerts: (listof Sym) \rightarrow Nat

(check-expect (count-concerts empty) 0)

(check-expect (count-concerts (cons 'a (cons 'b empty)))) 2)

(define (count-concerts los)

(cond [(empty? los) 0]

[else (+ 1 (count-concerts (rest los)))]))

This is a **recursive** function (it uses **recursion**).

A function is recursive when the body of the function involves an application of the same function.

This is an important technique which we will use quite frequently throughout the course.

Fortunately, our substitution rules allow us to trace such a function without much difficulty.

Tracing count-concerts

(count-concerts (cons 'a (cons 'b empty)))

⇒ (cond [(empty? (cons 'a (cons 'b empty))) 0]

[else (+ 1 (count-concerts (rest (cons 'a (cons 'b empty))))))])

⇒ (cond [false 0]

[else (+ 1 (count-concerts (rest (cons 'a (cons 'b empty))))))])

⇒ (cond [else (+ 1 (count-concerts (rest (cons 'a (cons 'b empty))))))])

⇒ (+ 1 (count-concerts (rest (cons 'a (cons 'b empty)))))

⇒ (+ 1 (count-concerts (cons 'b empty)))

$\Rightarrow (+\ 1\ (\text{cond}\ [(empty?\ (\text{cons}\ 'b\ \text{empty}))\ 0][\text{else}\ (+\ 1\ \dots)]))$
 $\Rightarrow (+\ 1\ (\text{cond}\ [\text{false}\ 0][\text{else}\ (+\ 1\ \dots)]))$
 $\Rightarrow (+\ 1\ (\text{cond}\ [\text{else}\ (+\ 1\ \dots)]))$
 $\Rightarrow (+\ 1\ (+\ 1\ (\text{count-concerts}\ (\text{rest}\ (\text{cons}\ 'b\ \text{empty}))))$
 $\Rightarrow (+\ 1\ (+\ 1\ (\text{count-concerts}\ \text{empty})))$
 $\Rightarrow (+\ 1\ (+\ 1\ (\text{cond}\ [(empty?\ \text{empty})\ 0][\text{else}\ (+\ 1\ \dots)])))$
 $\Rightarrow (+\ 1\ (+\ 1\ (\text{cond}\ [\text{true}\ 0][\text{else}\ (+\ 1\ \dots)])))$
 $\Rightarrow (+\ 1\ (+\ 1\ 0)) \Rightarrow (+\ 1\ 1) \Rightarrow 2$

Here we have used an omission ellipsis to avoid overflowing the slide.

Condensed traces

The full trace contains too much detail, so we instead use a **condensed trace** of the recursive function. This shows the important steps and skips over the trivial details.

This is a space saving tool we use in these slides, not a rule that you have to understand.

The condensed trace of our example

`(count-concerts (cons 'a (cons 'b empty)))`

\Rightarrow `(+ 1 (count-concerts (cons 'b empty)))`

\Rightarrow `(+ 1 (+ 1 (count-concerts empty)))`

\Rightarrow `(+ 1 (+ 1 0))`

\Rightarrow `2`

This condensed trace shows more clearly 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.

From now on, for the sake of readability, we will tend to use condensed traces. At times we will condense even more (for example, not fully expanding constants).

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

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.

Example: count-waterboys

:: (count-waterboys los) produces the number of occurrences

:: of 'Waterboys in los

:: count-waterboys: (listof Sym) \rightarrow Nat

:: Examples:

(check-expect (count-waterboys empty) 0)

(check-expect (count-waterboys (cons 'Waterboys empty)) 1)

(check-expect (count-waterboys (cons 'DaCapo
(cons 'U2 empty)))) 0)

(define (count-waterboys los) ...)

The template is a good place to start writing code. Write the template. Then, alter it according to the specific function you want to write.

For instance, we can generalize `count-waterboys` to a function which also consumes the symbol to be counted.

```
:: count-symbol: Sym (listof Sym) → Nat  
(define (count-symbol s los) ...)
```

The recursive function application will be `(count-symbol s (rest los))`.

Structural recursion

The template we have derived has the property that the form of the code matches the form of the data definition.

This type of recursion is known as **structural recursion**.

There are other types of recursion which we will see later on in the course.

Until we do, it is a good idea to keep in mind that the functions we write will use structural recursion (and hence will fit the form described by such templates).

Use the templates.

Design recipe refinements

The design recipe for functions involving self-referential data definitions generalizes this example.

Do this *once per self-referential data type*:

Data Analysis and Definition: This part of the design recipe will 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.

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.

cond-clauses corresponding to compound data clauses in the definition contain selector expressions.

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

The (listof Sym) template revisited

:: A (listof Sym) is one of:

:: ★ empty

:: ★ (cons Sym (listof Sym))

:: my-los-fn: (listof Sym) \rightarrow Any

(define (my-los-fn los)

(cond [(empty? los) ...]

[else (... (first los) ... (my-los-fn (rest los)) ...)]))

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

:: A (listof Num) is one of:

:: ★ empty

:: ★ (cons Num (listof Num))

:: my-lon-fn: (listof Num) \rightarrow Any

(define (my-lon-fn lon)

(cond [(empty? lon) ...]

[else (... (first lon) ... (my-lon-fn (rest lon)) ...)]))

Templates

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

`:: my-listof-X-fn: (listof X) → Any`

```
(define (my-listof-X-fn lst)
  (cond [(empty? lst) ...]
        [else (... (first lst)... (my-listof-X-fn (rest lst))...)]))
```

You will use this template *many* times in CS135!

Templates as generalizations

A template provides the basic shape of the code as suggested by the data definition.

Later in the course, we will learn about an abstraction mechanism (higher-order functions) that can reduce the need for templates.

We will also discuss alternatives for tasks where the basic shape provided by the template is not right for a particular computation.

Filling in the templates

In the Function Definition part of the design recipe: First write the **cond**-answers corresponding to base cases (which don't involve recursion).

For self-referential or recursive cases, figure out how to combine the values provided by the recursive application(s) and the selector expressions.

As always, create examples that exercise all parts of the data definition, and tests that exercise all parts of the code.

(Pure) structural recursion

In (pure) structural recursion, all arguments to the recursive function application (or applications, if there are more than one) are either:

- unchanged, or
- *one step* closer to a base case according to the data definition

Useful list functions

A closer look at `count-concerts` reveals that it will work just fine on any list.

In fact, it is a built-in function in Racket, under the name `length`.

Another useful built-in function is `member?`, which consumes an element of any type and a list, and returns `true` if the element is in the list, or `false` if it is not present.

Producing lists from lists

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

`:: (negate-list lon) produces a list with every number in lon negated`

`:: negate-list: (listof Num) \rightarrow (listof Num)`

`(check-expect (negate-list empty) empty)`

`(check-expect (negate-list (cons 2 (cons -12 empty)))
 (cons -2 (cons 12 empty)))`

Since `negate-list` consumes a `(listof Num)`, we use the general list template to write it.

negate-list with template

:: (negate-list lon) produces a list with every number in lon negated

:: negate-list: (listof Num) \rightarrow (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 lon)  
  (cond [(empty? lon) ...]  
        [else (... (first lon) ... (negate-list (rest lon)) ... )]))
```

negate-list completed

;; (negate-list lon) produces a list with every number in lon negated

;; negate-list: (listof Num) \rightarrow (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 lon)
  (cond [(empty? lon) empty]
        [else (cons (— (first lon)) (negate-list (rest lon)))]))
```


A condensed trace

```
(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))
```

Nonempty lists

Sometimes a given computation only makes sense on a nonempty list — for instance, finding the maximum of a list of numbers.

Exercise: create a data definition for a nonempty list of numbers, and use it to develop a template for functions that consume nonempty lists. Then write a function to find the maximum of a nonempty list.

Contracts for nonempty lists

Recall that we use `(listof X)` in contracts to refer to a list of elements of type `X`.

To emphasize the additional *requirement* that the list must be nonempty, we add a **requires** section.

For example, the function in the previous exercise (`max-list`) would have the following design recipe components:

`:: (max-list lon) produces the maximum element of lon`

`:: max-list: (listof Num) \rightarrow Num`

`:: requires: lon is nonempty`

Strings and lists of characters

Processing text is an extremely common task for computer programs. Text is usually represented in a computer by strings.

In Racket (and in many other languages), a string is really a sequence of **characters** in disguise.

Racket provides the function `string→list` to convert a string to a list of characters.

The function `list→string` does the reverse: it converts a list of characters into a string.

Racket's notation for the character 'a' is `#\a`.

The result of evaluating `(string→list "test")` is the list `(cons #\t (cons #\e (cons #\s (cons #\t empty))))`.

This is unfortunately ugly, but the `#` notation is part of a more general way of specifying values in Racket.

We have seen some of this, for example `#t` and `#f`, and we will see more in CS 136.

Most functions you write that consume or produce lists will use a **wrapper function**. Here is the **main function**:

```
:: (count-e/list loc) counts the number of occurrences of #\e in loc  
:: count-e/list: (listof Char) → Nat  
(check-expect (count-e/list empty) 0)  
(check-expect (count-e/list (cons #\a (cons #\e empty))) 1)  
(define (count-e/list loc)  
  ...)
```

Here is the **wrapper function**:

:: (count-e str) counts the number of occurrences of the letter e in str

:: count-e: Str \rightarrow Nat

:: Examples:

(check-expect (count-e " ") 0)

(check-expect (count-e "realize") 2)

```
(define (count-e str)
  (count-e/list (string $\rightarrow$ list str)))
```

Lists of structures

To write a function that consumes a list of structures, we use both the structure template and the list template.

Consider the following salary record structure:

```
(define-struct sr (name salary))
```

```
:: A Salary Record (SR) is a (make-sr Str Num)
```

```
:: my-sr-fn: SR → Any
```

```
(define (my-sr-fn sr)
```

```
  (... (sr-name sr) ...
```

```
    (sr-salary sr) ... ))
```


The following is a template function for a list of salary records.

:: my-listof-sr-fn: (listof SR) \rightarrow Any

```
(define (my-listof-sr-fn lst)
  (cond [(empty? lst) ...]
        [else (... (my-sr-fn (first lst)) ...
                    (my-listof-sr-fn (rest lst)) ... )]))
```

Because we know that `(first lst)` is a Salary Record, it suggests using our salary record template function.

An alternative is to integrate the two templates into a single template.

```
(define (my-listof-sr-fn lst)
  (cond [(empty? lst) ...]
        [else (... (sr-name (first lst)) ...
                     (sr-salary (first lst)) ...
                     (my-listof-sr-fn (rest lst)) ... )]))
```

The approach you use is a matter of judgment.

In the following example, we will use two separate functions.

Example: compute-taxes We want to write a function `compute-taxes` that consumes a list of salary records and produces a list of corresponding tax records.

```
(define-struct tr (name tax))
```

```
:: A Tax Record (TR) is a (make-tr Str Num)
```

To determine the tax amount, we use our `tax-payable` function from module 02.

```
(define srlst (cons (make-sr "Jane Doe" 50000)
                    (cons (make-sr "Da Kou" 15500)
                          (cons (make-sr "MusaAlKhwarizmi" 100000) empty))))
```

:: (compute-taxes lst) produces a list of tax records,

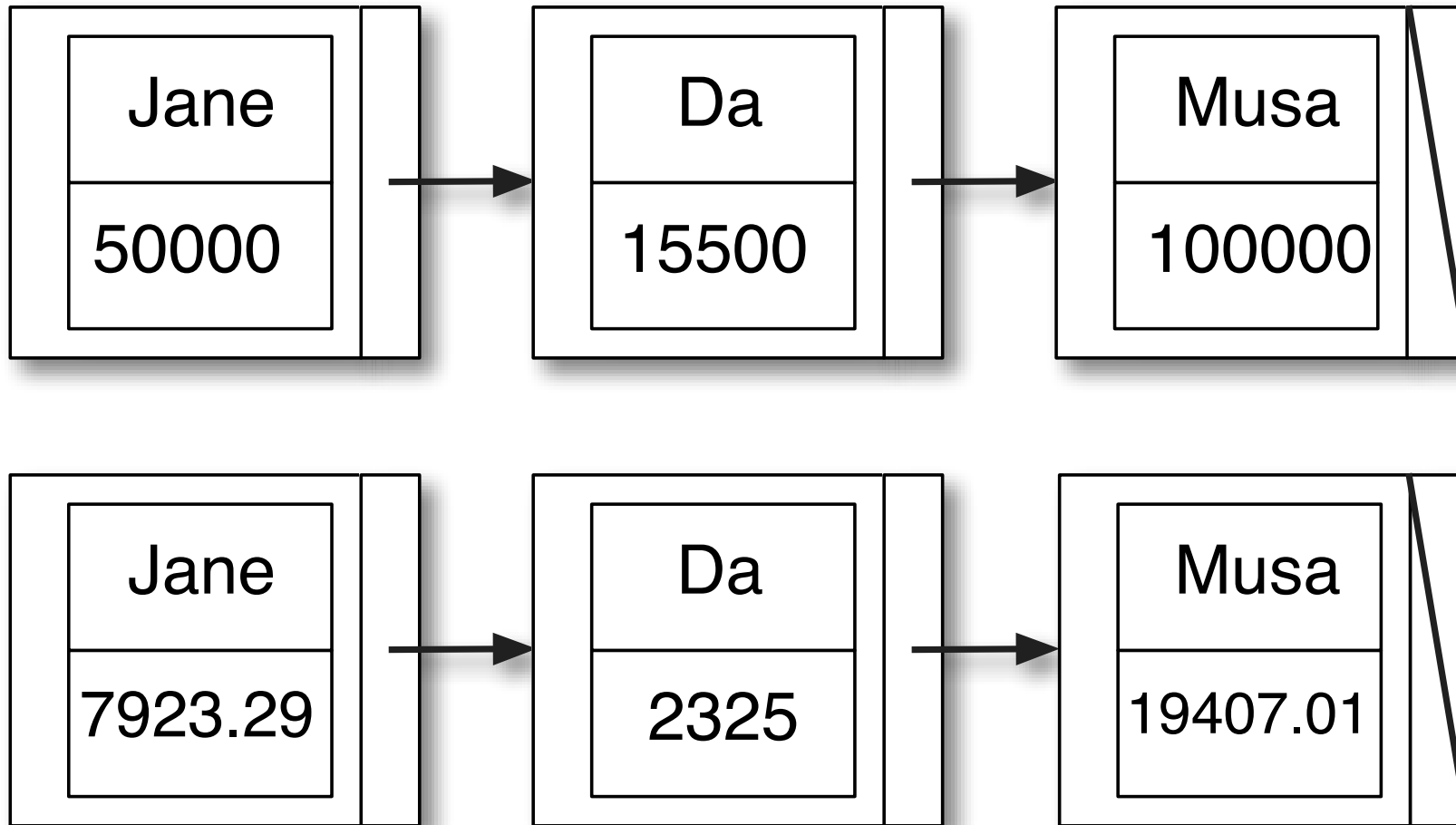
:: one for each salary record in lst

:: compute-taxes: (listof SR) \rightarrow (listof TR)

:: Example:

```
(check-expect (compute-taxes srlst)
              (cons (make-tr "Jane Doe" 7923.29)
                    (cons (make-tr "Da Kou" 2325)
                          (cons (make-tr "MusaAlKhwarizmi" 19407.01) empty))))
```

Envisioning compute-taxes



```
(define srlst (cons (make-sr "Jane Doe" 50000)
                    (cons (make-sr "Da Kou" 15500)
                          (cons (make-sr "MusaAlKhwarizmi" 100000) empty))))
```

What do these evaluate to?

- (first srlst) ?
- (sr-salary (first srlst)) ?
- (rest srlst) ?

The function compute-taxes

The base case is easy: an empty list produces an empty list.

`[(empty? lst) empty]`

For the self-referential case, `(first lst)` is a salary record.

From it, we can compute a tax record, which should go at the front of the list of tax records produced.

We do this with the helper function `sr→tr`.

In other words, to produce the answer, `(sr→tr (first lst))` should be `consed` onto `(compute-taxes (rest lst))`.

:: (compute-taxes lst) produces a list of tax records,

:: one for each salary record in lst

:: compute-taxes: (listof SR) \rightarrow (listof TR)

```
(define (compute-taxes lst)
  (cond [(empty? lst) empty]
        [else (cons (sr $\rightarrow$ tr (first lst))
                      (compute-taxes (rest lst)))]))
```

The function `sr \rightarrow tr` uses the SR template (plus our previously-written function `tax-payable`).

:: (sr→tr sr) produces a tax record for sr

:: sr→tr: SR → TR

:: Example:

```
(check-expect (sr→tr (make-sr "Jane Doe" 50000))  
              (make-tr "Jane Doe" 7923.29))
```

```
(define (sr→tr sr)  
  (make-tr  
    (sr-name sr)  
    (tax-payable (sr-salary sr))))
```

A condensed trace

(compute-taxes srlst)

⇒ (compute-taxes

 (cons (make-sr "Jane Doe" 50000)

 (cons (make-sr "Da Kou" 15500)

 (cons (make-sr "MusaAlKhwarizmi" 100000) empty))))

⇒ (cons (make-tr "Jane Doe" 7923.29)

 (compute-taxes

 (cons (make-sr "Da Kou" 15500)

 (cons (make-sr "MusaAlKhwarizmi" 100000) empty))))

```
⇒ (cons (make-tr "Jane Doe" 7923.29)
      (cons (make-tr "Da Kou" 2325)
            (compute-taxes
              (cons (make-sr "MusaAlKhwarizmi" 100000) empty))))
⇒ (cons (make-tr "Jane Doe" 7923.29)
      (cons (make-tr "Da Kou" 2325)
            (cons (make-tr "MusaAlKhwarizmi" 19407.01)
                  (compute-taxes empty))))
⇒ (cons (make-tr "Jane Doe" 7923.29)
      (cons (make-tr "Da Kou" 2325)
            (cons (make-tr "MusaAlKhwarizmi" 19407.01) empty)))
```

Goals of this module

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

You should understand box-and-pointer visualization of lists.

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

You should understand and use (`listof . . .`) notation in contracts.

You should understand strings, their relationship to characters and how to convert a string into a list of characters (and vice-versa).

You should be comfortable with lists of structures, including understanding the recursive definitions of such data types, and you should be able to derive and use a template based on such a definition.

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.

We also look at function which do recursion on more than one item (lists and/or numbers) simultaneously.

From definition to template

We'll review how we derived the list template.

:: A List is one of:

:: ★ `empty`

:: ★ `(cons Any List)`

Suppose we have a list `lst`.

The test `(empty? lst)` tells us which case applies.

If `(empty? lst)` is `false`, then `lst` is of the form `(cons f r)`.

How do we compute the values `f` and `r`?

`f` is `(first lst)`.

`r` is `(rest lst)`.

Because `r` is a list, we recursively apply the function we are constructing to it.

:: my-list-fn: (listof Any) \rightarrow Any

```
(define (my-list-fn lst)
  (cond [(empty? lst) ...]
        [else (... (first lst) ...
                     (my-list-fn (rest lst)) ... )]))
```

We can repeat this reasoning on a recursive definition of natural numbers to obtain a template.

Natural numbers

:: A Nat is one of:

:: ★ 0

:: ★ (add1 Nat)

Here `add1` is the built-in function that adds 1 to its argument.

The natural numbers start at 0 in computer science and some branches of mathematics (e.g. logic).

We'll now work out a template for functions that consume a natural number.

Suppose we have a natural number n .

The test `(zero? n)` tells us which case applies.

If `(zero? n)` is `false`, then n has the value `(add1 k)` for some k .

To compute k , we subtract 1 from n , using the built-in `sub1` function.

Because the result `(sub1 n)` is a natural number, we recursively apply the function we are constructing to it.

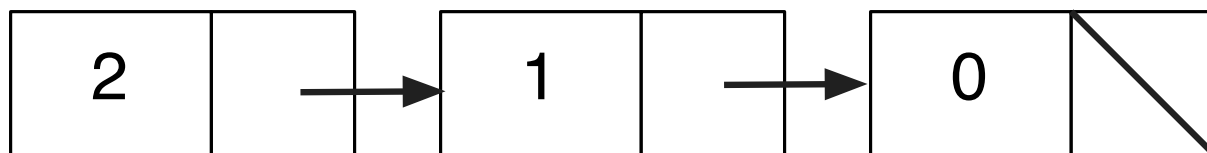
```
(define (my-nat-fn n)
  (cond [(zero? n) ...]
        [else (... (my-nat-fn (sub1 n)) ...)]))
```

Example: a decreasing list

Goal: `countdown`, which consumes a natural number n and produces a decreasing list of all natural numbers less than or equal to n .

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

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



We start filling in the template:

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

If n is 0, we produce the list containing 0, and if n is nonzero, we cons n onto the countdown list for $n - 1$.

:: (countdown n) produces a list of Nats from n...0

:: countdown: Nat \rightarrow (listof Nat)

:: Example:

```
(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)))]))
```

A condensed trace

(countdown 2)

⇒ (cons 2 (countdown 1))

⇒ (cons 2 (cons 1 (countdown 0)))

⇒ (cons 2 (cons 1 (cons 0 empty)))

If the function `countdown` is applied to a negative integer, it will not terminate. Even though we make no guarantees when the contract is violated, we should program “defensively” where possible.

We could check for this and signal an error, or we could change the test to catch this (and some other situations).

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


Some useful notation

The symbol \mathbb{Z} is often used to denote the integers.

We can add subscripts to define subsets of the integers.

For example, $\mathbb{Z}_{\geq 0}$ defines the non-negative integers, also known as the natural numbers.

Other examples: $\mathbb{Z}_{>4}$, $\mathbb{Z}_{<-8}$, $\mathbb{Z}_{\leq 1}$.

Subintervals of the natural numbers

If we change the base case test from `(zero? n)` to `(= n 7)`, we can stop the countdown at 7.

This corresponds to the following definition:

:: An integer in $\mathbb{Z}_{\geq 7}$ is one of:

:: $\star 7$

:: $\star (\text{add1 } \mathbb{Z}_{\geq 7})$

(In practice, we add a require section to our contract.)

:: (countdown-to-7 n) produces a decreasing list from n...7

:: countdown-to-7: Nat \rightarrow (listof Nat)

:: requires: $n \geq 7$

:: Example:

(check-expect (countdown-to-7 9) (cons 9 (cons 8 (cons 7 empty))))

```
(define (countdown-to-7 n)
  (cond [(= n 7) (cons 7 empty)]
        [else (cons n (countdown-to-7 (sub1 n)))]))
```

Again, making the base case be $(\leq n 7)$ is more robust.

We can generalize both `countdown` and `countdown-to-7` by providing the base value (e.g. 0 or 7) as a second parameter `b` (the “base”).

Here, the stopping condition will depend on `b`.

The parameter `b` has to go “along for the ride” in the recursion.

:: (countdown-to n b) produces a list from n...b

:: countdown-to: Int Int \rightarrow (listof Int)

:: requires: $n \geq b$

:: Example:

(check-expect (countdown-to 4 2) (cons 4 (cons 3 (cons 2 empty))))

(define (countdown-to n b)

(cond [(= n b) (cons b empty)]

[else (cons n (countdown-to (sub1 n) b))]))

Another condensed trace

(countdown-to 4 2)

⇒ (cons 4 (countdown-to 3 2))

⇒ (cons 4 (cons 3 (countdown-to 2 2)))

⇒ (cons 4 (cons 3 (cons 2 empty)))

`countdown-to` works just fine if we put in negative numbers.

```
(countdown-to 1 -2)
```

```
⇒ (cons 1 (cons 0 (cons -1 (cons -2 empty))))
```

Here is the template for counting down to `b`.

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

Going the other way

What if we want an increasing count?

Consider the non-positive integers $\mathbb{Z}_{\leq 0}$.

:: A integer in $\mathbb{Z}_{\leq 0}$ is one of:

:: $\star 0$

:: $\star (\text{sub1 } \mathbb{Z}_{\leq 0})$

Examples: -1 is $(\text{sub1 } 0)$, -2 is $(\text{sub1 } (\text{sub1 } 0))$.

If an integer i is of the form $(\text{sub1 } k)$, then k is equal to $(\text{add1 } i)$. This suggests the following template.

Notice the additional requires section.

```
:: my-nonpos-fn: Int  $\rightarrow$  Any
```

```
:: requires: n  $\leq$  0
```

```
(define (my-nonpos-fn n)  
  (cond [(zero? n) ...]  
        [else (... (my-nonpos-fn (add1 n)) ...)]))
```

We can use this to develop a function to produce lists such as

```
(cons -2 (cons -1 (cons 0 empty))).
```

:: (countup n) produces a list from n...0

:: countup: Int \rightarrow (listof Int)

:: requires: $n \leq 0$

:: Example:

(check-expect (countup -2) (cons -2 (cons -1 (cons 0 empty))))

(define (countup n)

(cond [(zero? n) (cons 0 empty)]

[else (cons n (countup (add1 n)))]))

As before, we can generalize this to counting up to b , by introducing b as a second parameter in a template.

```
(define (my-upto-b-fn n b)
  (cond [(= n b) (... b ...)]
        [else (... (my-upto-b-fn (add1 n) b) ... )]))
```

:: (countup-to n b) produces a list from n...b

:: countup-to: Int Int \rightarrow (listof Int)

:: requires: $n \leq b$

:: Example:

(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))]))

Yet another condensed trace

(countup-to 6 8)

⇒ (cons 6 (countup-to 7 8))

⇒ (cons 6 (cons 7 (countup-to 8 8)))

⇒ (cons 6 (cons 7 (cons 8 empty)))

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

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

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

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

When you are learning to use recursion, 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 using the right pattern.

- ★ You may **not** use `reverse` on assignments unless we say otherwise.

More complicated situations

As before, we may need to introduce auxiliary functions during the composition of a function. These may or may not be recursive themselves.

Sorting a list of numbers provides a good example; in this case the solution follows easily from the templates and design process.

In this course and CS 136, we will see several different sorting algorithms.

Filling in the list template

:: (sort lon) sorts the elements of lon in nondecreasing order

:: sort: (listof Num) \rightarrow (listof Num)

```
(define (sort lon)
  (cond [(empty? lon) ...]
        [else (... (first lon) ...
                     (sort (rest lon)) ... )]))
```

If the list `lon` is empty, so is the result.

Otherwise, the template suggests doing something with the first element of the list, and the sorted version of the rest.

```
(define (sort lon)
  (cond [(empty? lon) empty]
        [else (insert (first lon) (sort (rest lon)))]))
```

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

A condensed trace of **sort** and **insert**

```
(sort (cons 2 (cons 4 (cons 3 empty))))  
⇒ (insert 2 (sort (cons 4 (cons 3 empty))))  
⇒ (insert 2 (insert 4 (sort (cons 3 empty))))  
⇒ (insert 2 (insert 4 (insert 3 (sort empty))))  
⇒ (insert 2 (insert 4 (insert 3 empty)))  
⇒ (insert 2 (insert 4 (cons 3 empty)))  
⇒ (insert 2 (cons 3 (cons 4 empty)))  
⇒ (cons 2 (cons 3 (cons 4 empty)))
```

The auxiliary function **insert**

We again use the list template for **insert**.

:: (insert n slon) inserts the number n into the sorted list slon

:: so that the resulting list is also sorted.

:: insert: Num (listof Num) \rightarrow (listof Num)

:: requires: slon is sorted in nondecreasing order

```
(define (insert n slon)
  (cond [(empty? slon) ...]
        [else (... (first slon) ...
                     (insert n (rest slon)) ... )]))
```

If `slon` is empty, the result is the list containing just `n`.

If `slon` is not empty, another conditional expression is needed.

`n` is the first number in the result if it is less than or equal to the first number in `slon`.

Otherwise, the first number in the result is the first number in `slon`, and the rest of the result is what we get when we insert `n` into `(rest slon)`.

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

A condensed trace of **insert**

```
(insert 4 (cons 1 (cons 2 (cons 5 empty))))  
⇒ (cons 1 (insert 4 (cons 2 (cons 5 empty))))  
⇒ (cons 1 (cons 2 (insert 4 (cons 5 empty))))  
⇒ (cons 1 (cons 2 (cons 4 (cons 5 empty))))|
```

Our **sort** with helper function **insert** are together known as **insertion sort**.

List abbreviations

Now that we understand 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 1 2 4 5).
```


Beginning Student With List Abbreviations also provides some shortcuts for accessing specific elements of lists.

Recall slide 01-23 to change language in DrRacket.

(`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.

The templates we have developed remain very useful.

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 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).

Quoting lists

If lists built using `list` consist of just symbols, strings, and numbers, the list abbreviation can be further abbreviated using the quote notation we used for symbols.

`(cons 'red (cons 'blue (cons 'green empty)))` can be written `'(red blue green)`.

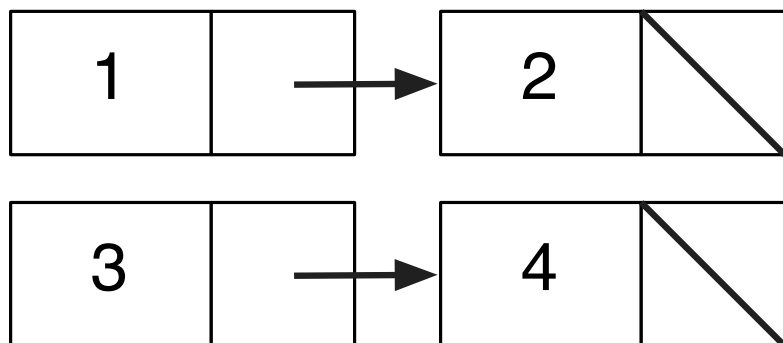
`(list 5 4 3 2)` can be written `'(5 4 3 2)`, because quoted numbers evaluate to numbers; that is, `'1` is the same as `1`.

What is `'()` ?

Lists containing lists

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

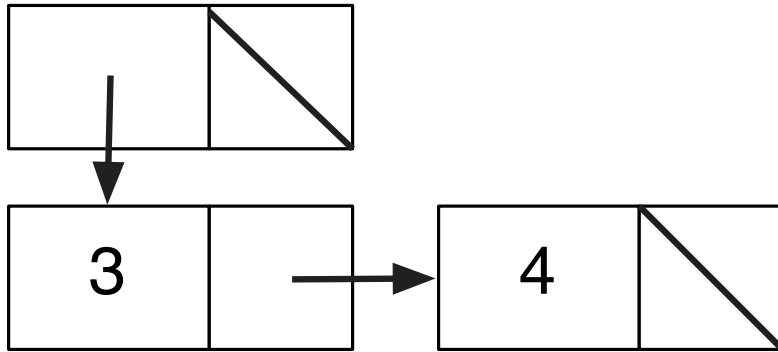
Here are two different two-element lists.



```
(cons 1 (cons 2 empty))
```

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

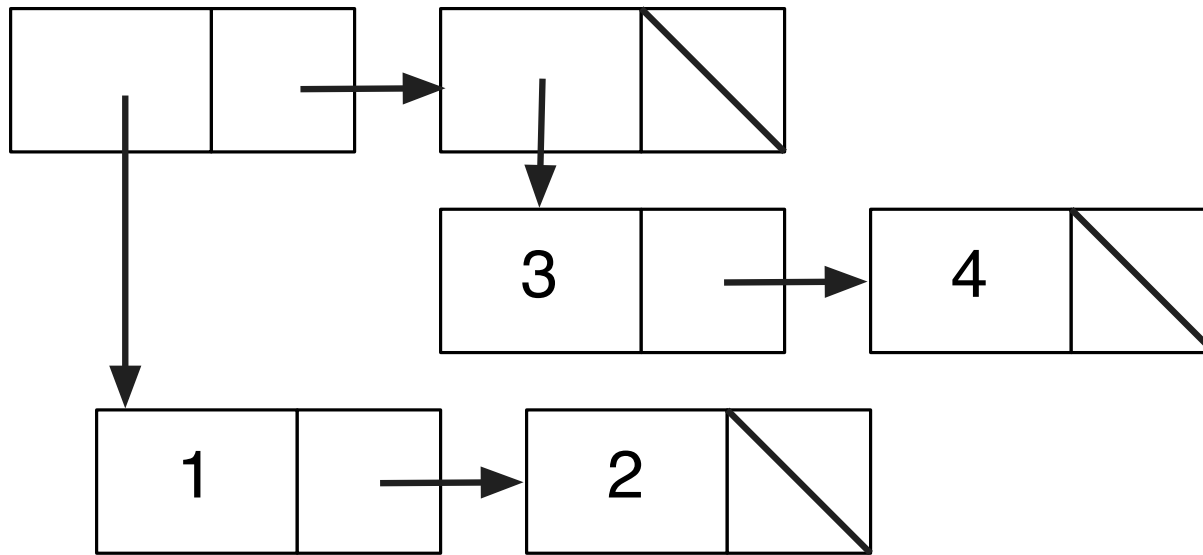
Here is a one-element list whose single element is one of the two-element lists we saw above.



```
(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))
```

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))
```

```
'((1 2) (3 4))
```

Clearly, the abbreviations are more expressive.

Lists versus structures

Since lists can contain lists, they offer an alternative to using structures.

Previously, we defined a salary record as a two-field structure but we could have defined a salary record as a two element list.

:: An Alternative Salary Record (AltSR) is a (list Str Num)

```
(list (list "Jane Doe" 50000)
      (list "Da Kou" 15500)
      (list "MuzaAlKhwarismi" 100000))
```


We can use our AltSR data definition to generate a template function, and for a list of AltSR, we can combine the templates.

;; An Alternative Salary Record (AltSR) is a (list Str Num)

;; my-altsr-fn: (list Str Num) \rightarrow Any

(define (my-altsr-fn asr)

(... (first asr) ... (second asr) ...))

;; my-listof-altsr-fn: (listof (list Str Num)) \rightarrow Any

(define (my-listof-altsr-fn lst)

(cond [(empty? lst) ...]

[else (... (first (first lst)) ... ; name of first

(second (first lst)) ... ; salary of first

(my-listof-altsr-fn (rest lst)) ...]))

Example: a function `name-list` which consumes a list of AltSR and produces the corresponding list of names.

`:: name-list: (listof (list Str Num)) \rightarrow (listof Str)`

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

This code is less readable, because it uses only lists, instead of structures, and so is more generic-looking.

We can fix this with a few definitions.

```
(define (name x) (first x))
```

```
(define (salary x) (second x))
```

:: name-list: (listof (list Str Num)) \rightarrow (listof Str)

```
(define (name-list lst)
```

```
  (cond [(empty? lst) empty]
```

```
        [else (cons (name (first lst))
```

```
                    (name-list (rest lst))))]))
```

This is one of the ways that structures can be simulated. There are others, as we will see.

Why use lists containing lists?

We could reuse the `name-list` function to produce a list of names from an list of tax records.

Our original structure-based salary record and tax record definitions would require two different (but very similar) functions.

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

Why use structures?

Structure is often present in a computational task, or can be defined to help handle a complex situation.

Using structures helps avoid some programming errors (e.g., accidentally extracting a list of salaries instead of names).

Our design recipes can be adapted to give guidance in writing functions using complicated structures.

Most mainstream programming languages provide structures for use in larger programming tasks.

Some statically typed programming languages provide mechanisms that avoid having to duplicate code for types that are structurally identical or similar.

Inheritance in object-oriented languages is one such mechanism.

Such features could be easily built into the teaching languages, but are not, because the goal is to make a transition to other languages.

The full version of Racket (which goes beyond the standard, as we will see in CS 136) does have such features.

Different kinds of lists

When we introduced lists in module 05, the items they contained were not lists. These were **flat lists**.

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

In later lecture modules, we will use lists containing unbounded flat lists.

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

Dictionaries

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

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

Example: the telephone directory. Keys are names, and values are telephone numbers.

Example: your seat assignment for midterms. Keys are userids, and values are seat locations.

More generally, any two-column table can be viewed as a dictionary.

Example: your report card, where the first column contains courses, and the second column contains corresponding marks.

What *operations* might we wish to perform on dictionaries?

- **lookup**: given key, produce corresponding value
- **add**: add a (key,value) pair to the dictionary
- **remove**: given key, remove it and associated value

Association lists

One simple solution uses an **association list**, which is just a list of (key, value) pairs.

We store the pair as a two-element list. For simplicity, we will make the keys numbers, and the values strings.

:: An association list (AL) is one of:

:: ★ *empty*

:: ★ (cons (list Num Str) AL)

We can, in an association list, use other types for keys and values. We have used `Num` and `Str` here just to provide a concrete example. We impose the additional restriction that an association list contains at most one occurrence of any key. Since we have a data definition, we could use `AL` for the type of an association list, as given in a contract. Another alternative is to use `(listof (list Num Str))`.

We can use the data definition to produce a template.

:: my-al-fn: AL \rightarrow Any

```
(define (my-al-fn alst)
  (cond [(empty? alst) ...]
        [else (... (first (first alst)) ... ; first key
                     (second (first alst)) ... ; first value
                     (my-al-fn (rest alst))))]))
```

In coding the lookup function, we have to make a decision. What should we produce if the lookup fails?

Since all valid values are strings, we can produce `false` to indicate that the key was not present in the association list.

:: (lookup-al k alst) produces the value corresponding to key k,
:: or false if k not present
:: lookup-al: Num AL \rightarrow (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.

In a future module, we will impose structure to improve this situation.

Two-dimensional data

Another use of lists of lists is to represent a two-dimensional table.

For example, here is a multiplication table:

`(mult-table 3 4) ⇒`

`(list (list 0 0 0 0)`

`(list 0 1 2 3)`

`(list 0 2 4 6))`

The c^{th} entry of the r^{th} row (numbering from 0) is $r * c$.

We can write `mult-table` using two applications of the “count up” idea.

:: (mult-table nr nc) produces multiplication table

:: with nr rows and nc columns

:: mult-table: Nat Nat \rightarrow (listof (listof Nat))

```
(define (mult-table nr nc)
```

```
  (rows-from 0 nr nc))
```

:: (rows-from r nr nc) produces mult. table, rows r...(nr-1)

:: rows-from: Nat Nat Nat \rightarrow (listof (listof Nat))

```
(define (rows-from r nr nc)
```

```
  (cond [( $\geq$  r nr) empty]
```

```
        [else (cons (row r nc) (rows-from (add1 r) nr nc))]))
```

:: (row r nc) produces rth row of mult. table of length nc

:: row: Nat Nat \rightarrow (listof Nat)

```
(define (row r nc)
  (cols-from 0 r nc))
```

:: (cols-from c r nc) produces entries c...(nc-1) of rth row of mult. table

:: cols-from: Nat Nat Nat \rightarrow (listof Nat)

```
(define (cols-from c r nc)
  (cond [( $\geq$  c nc) empty]
        [else (cons (* r c) (cols-from (add1 c) r nc))]))
```

Processing two lists simultaneously

We now skip ahead to section 17 material.

This involves more complicated recursion, namely in writing functions which consume two lists (or two data types, each of which has a recursive definition).

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

The simplest case is when one of the lists does not require recursive processing.

Case 1: processing just one list

As an example, consider the function `my-append`.

`:: (my-append lst1 lst2) appends lst2 to the end of lst1`

`:: my-append: (listof Any) (listof Any) \rightarrow (listof Any)`

`:: Examples:`

`(check-expect (my-append empty '(1 2)) '(1 2))`

`(check-expect (my-append '(3 4) '(1 2 5)) '(3 4 1 2 5))`

```
(define (my-append lst1 lst2)
  ...)
```

```
(define (my-append lst1 lst2)
  (cond [(empty? lst1) lst2]
        [else (cons (first lst1)
                      (my-append (rest lst1) lst2))]))
```

The code only does structural recursion on `lst1`.

The parameter `lst2` is “along for the ride”.

A condensed trace

```
(my-append (cons 1 (cons 2 empty)) (cons 3 (cons 4 empty)))  
⇒ (cons 1 (my-append (cons 2 empty) (cons 3 (cons 4 empty))))  
⇒ (cons 1 (cons 2 (my-append empty (cons 3 (cons 4 empty)))))  
⇒ (cons 1 (cons 2 (cons 3 (cons 4 empty))))
```

Case 2: processing in lockstep

To process two lists `lst1` and `lst2` in lockstep, they must be the same length, and are consumed at the same rate.

`lst1` is either `empty` or a `cons`, and the same is true of `lst2` (four possibilities in total).

However, because the two lists must be the same length, `(empty? lst1)` is `true` if and only if `(empty? lst2)` is `true`.

This means that out of the four possibilities, two are invalid for proper data.

The template is thus simpler than in the general case.

```
(define (my-lockstep-fn lst1 lst2)
  (cond [(empty? lst1) ... ]
        [else
         (... (first lst1) ... (first lst2) ...
              (my-lockstep-fn (rest lst1) (rest lst2)) ... )]))
```


Example: dot product

To take the dot product of two vectors, we multiply entries in corresponding positions (first with first, second with second, and so on) and sum the results.

Example: the dot product of $(1\ 2\ 3)$ and $(4\ 5\ 6)$ is

$$1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32.$$

We can store the elements of a vector in a list, so $(1\ 2\ 3)$ becomes `'(1 2 3)`.

For convenience, we define the empty vector with no entries, represented by `empty`.

The function dot-product

:: (dot-product lon1 lon2) computes the dot product

:: of vectors lon1 and lon2

:: dot-product: (listof Num) (listof Num) \rightarrow Num

:: requires: lon1 and lon2 are the same length

(check-expect (dot-product empty empty) 0)

(check-expect (dot-product '(2) '(3)) 6)

(check-expect (dot-product '(2 3) '(4 5)) 23)

```
(define (dot-product lon1 lon2)  
  ...)
```

```
(define (dot-product lon1 lon2)
  (cond [(empty? lon1) 0]
        [else (+ (* (first lon1) (first lon2))
                   (dot-product (rest lon1) (rest lon2)))]))
```

A condensed trace

```
(dot-product (cons 2 (cons 3 empty))  
             (cons 4 (cons 5 empty)))  
⇒ (+ 8 (dot-product (cons 3 empty)  
                     (cons 5 empty)))  
⇒ (+ 8 (+ 15 (dot-product empty  
              empty)))  
⇒ (+ 8 (+ 15 0)) ⇒ 23
```

Case 3: processing at different rates

If the two lists `lst1`, `lst2` being consumed are of different lengths, all four possibilities for their being empty/nonempty are possible:

```
(and (empty? lst1) (empty? lst2))
```

```
(and (empty? lst1) (cons? lst2))
```

```
(and (cons? lst1) (empty? lst2))
```

```
(and (cons? lst1) (cons? lst2))
```

Exactly one of these is true, but all must be tested in the template.

The template so far

```
(define (my-twolist-fn lst1 lst2)
  (cond [(and (empty? lst1) (empty? lst2)) ...]
        [(and (empty? lst1) (cons? lst2)) ...]
        [(and (cons? lst1) (empty? lst2)) ...]
        [(and (cons? lst1) (cons? lst2)) ... ]))
```

The first case is a **base case**; the second and third may or may not be.

Refining the template

```
(define (my-twolist-fn lst1 lst2)
  (cond
    [(and (empty? lst1) (empty? lst2)) ...]
    [(and (empty? lst1) (cons? lst2)) (... (first lst2) ... (rest lst2) ...)]
    [(and (cons? lst1) (empty? lst2)) (... (first lst1) ... (rest lst1) ...)]
    [(and (cons? lst1) (cons? lst2)) ??? ]))
```

The second and third cases may or may not require recursion.

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

Further refinements

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

... (first lst2) ... (my-twolist-fn lst1 (rest lst2)) ...

... (first lst1) ... (my-twolist-fn (rest lst1) lst2) ...

... (first lst1) ... (first lst2) ... (my-twolist-fn (rest lst1) (rest lst2)) ...

We need to reason further in specific cases to determine which is appropriate.

Example: merging two sorted lists

We wish to design a function `merge` that consumes two lists.

Each list is sorted in ascending order (no duplicate values).

`merge` will produce one such list containing all elements.

As an 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) \Rightarrow empty`

`(merge empty`

`(cons 2 empty)) \Rightarrow (cons 2 empty)`

`(merge (cons 1 (cons 3 empty))`

`empty) \Rightarrow (cons 1 (cons 3 empty))`

`(merge (cons 1 (cons 4 empty))`

`(cons 2 empty)) \Rightarrow (cons 1 (cons 2 (cons 4 empty)))`

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

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

Reasoning about merge

If `lon1` and `lon2` are both nonempty, what is the first element of the merged list?

It is the smaller of `(first lon1)` and `(first lon2)`.

If `(first lon1)` is smaller, then the rest of the answer is the result of merging `(rest lon1)` and `lon2`.

If `(first lon2)` is smaller, then the rest of the answer is the result of merging `lon1` and `(rest lon2)`.

```
(define (merge lon1 lon2)
  (cond [(and (empty? lon1) (empty? lon2)) empty]
        [(and (empty? lon1) (cons? lon2)) lon2]
        [(and (cons? lon1) (empty? lon2)) lon1]
        [(and (cons? lon1) (cons? lon2))
         (cond [(< (first lon1) (first lon2))
                  (cons (first lon1) (merge (rest lon1) lon2))]
               [else (cons (first lon2) (merge lon1 (rest lon2)))]))]))
```

A condensed trace

(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 condensed trace (with lists)

(merge (list 3 4)

(list 2 5 6))

⇒ (cons 2 (merge (list 3 4)

(list 5 6))))

⇒ (cons 2 (cons 3 (merge (list 4)

(list 5 6))))

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

(list 5 6))))))

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

Consuming a list and a number

We defined recursion on natural numbers by showing how to view a natural number in a list-like fashion.

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

A predicate “Does *elem* appear at least *n* times in this list?”

Example: “Does 2 appear at least 3 times in the list (`list 4 2 2 3 2 4`)?” returns `true`.

The function at-least?

:: (at-least? n elem lst) determines if elem appears

:: at least n times in lst.

:: at-least?: Nat Any (listof Any) \rightarrow Bool

:: Examples:

(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 code

The recursion will involve the parameters `n` and `lst`, once again giving four possibilities:

```
(define (at-least? n elem lst)
  (cond [(and (zero? n) (empty? lst)) ...]
        [(and (zero? n) (cons? lst)) ...]
        [(and (> n 0) (empty? lst)) ...]
        [(and (> n 0) (cons? lst)) ...]))
```

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

Refining at-least?

```
(define (at-least? n elem lst)
  (cond [(and (zero? n) (empty? lst)) ...]
        [(and (zero? n) (cons? lst)) ... ]
        [(and (> n 0) (empty? lst)) ... ]
        [(and (> n 0) (cons? lst)) ???]))
```

In which cases can we return the answer without further processing?

In which cases do we need further recursive processing to discover the answer? Which of the natural recursions should be used?

Improving at-least?

In working out the details for each case, it becomes apparent that some of them can be combined.

If `n` is zero, it doesn't matter whether `lst` is `empty` or not. Logically, every element always appears at least 0 times.

This leads to some rearrangement of the template, and eventually to the code that appears on the next slide.

Improved at-least?

```
(define (at-least? n elem lst)
  (cond [(= n 0) true]
        [(empty? lst) false]
        ; list is nonempty,  $n \geq 1$ 
        [(equal? (first lst) elem) (at-least? (sub1 n) elem (rest lst))]
        [else (at-least? n elem (rest lst))]))
```

Two condensed traces

(at-least? 3 'green (list 'red 'green 'blue)) \Rightarrow

(at-least? 3 'green (list 'green 'blue)) \Rightarrow

(at-least? 2 'green (list 'blue)) \Rightarrow

(at-least? 2 'green empty) \Rightarrow false

(at-least? 1 8 (list 4 8 15 16 23 42)) \Rightarrow

(at-least? 1 8 (list 8 15 16 23 42)) \Rightarrow

(at-least? 0 8 (list 15 16 23 42)) \Rightarrow true

Testing list equality

:: (list=? lst1 lst2) determines if lst1 and lst2 are equal

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

```
(define (list=? lst1 lst2)
```

```
  (cond
```

```
    [(and (empty? lst1) (empty? lst2)) ...]
```

```
    [(and (empty? lst1) (cons? lst2)) (... (first lst2) ... (rest lst2) ...)]
```

```
    [(and (cons? lst1) (empty? lst2)) (... (first lst1) ... (rest lst1) ...)]
```

```
    [(and (cons? lst1) (cons? lst2)) ??? ]))
```

Reasoning about list equality

Two empty lists are equal; if one is empty and the other is not, they are not equal.

If both are nonempty, then their first elements must be equal, and their rests must be equal.

The natural recursion in this case is

```
(list=? (rest lst1) (rest lst2))
```

```
(define (list=? lst1 lst2)
  (cond [(and (empty? lst1) (empty? lst2)) true]
        [(and (empty? lst1) (cons? lst2)) false]
        [(and (cons? lst1) (empty? lst2)) false]
        [(and (cons? lst1) (cons? lst2))
         (and (= (first lst1) (first lst2))
               (list=? (rest lst1) (rest lst2))))]))
```

Some further simplifications are possible.

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 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 m , 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 and quote notation for lists where appropriate.

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

You should understand the similar uses of structures and fixed-size lists, and be able to write functions that consume either type of data.

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 suitable in a given situation.

Types of recursion

Readings: none.

In this module:

- a glimpse of non-structural recursion

Structural vs. general recursion

All of the recursion we have done to date has been **structural**.

The templates we have been using have been derived from a data definition and specify the form of the recursive application.

We will continue to use primarily structural recursion for several more lecture modules.

But here we'll take a brief peek ahead at more general forms of recursion.

(Pure) structural recursion

Recall from Module 05:

In (pure) structural recursion, all arguments to the recursive function application (or applications, if there are more than one) are either:

- unchanged, or
- *one step* closer to a base case according to the data definition

The limits of structural recursion

;; (max-list lon) produces the maximum element of lon

;; max-list: (listof Num) \rightarrow Num

;; requires: lon is nonempty

```
(define (max-list lon)
  (cond [(empty? (rest lon)) (first lon)]
        [(> (first lon) (max-list (rest lon))) (first lon)]
        [else (max-list (rest lon))]))
```

There may be two recursive applications of `max-list`.

The code for `max-list` is correct.

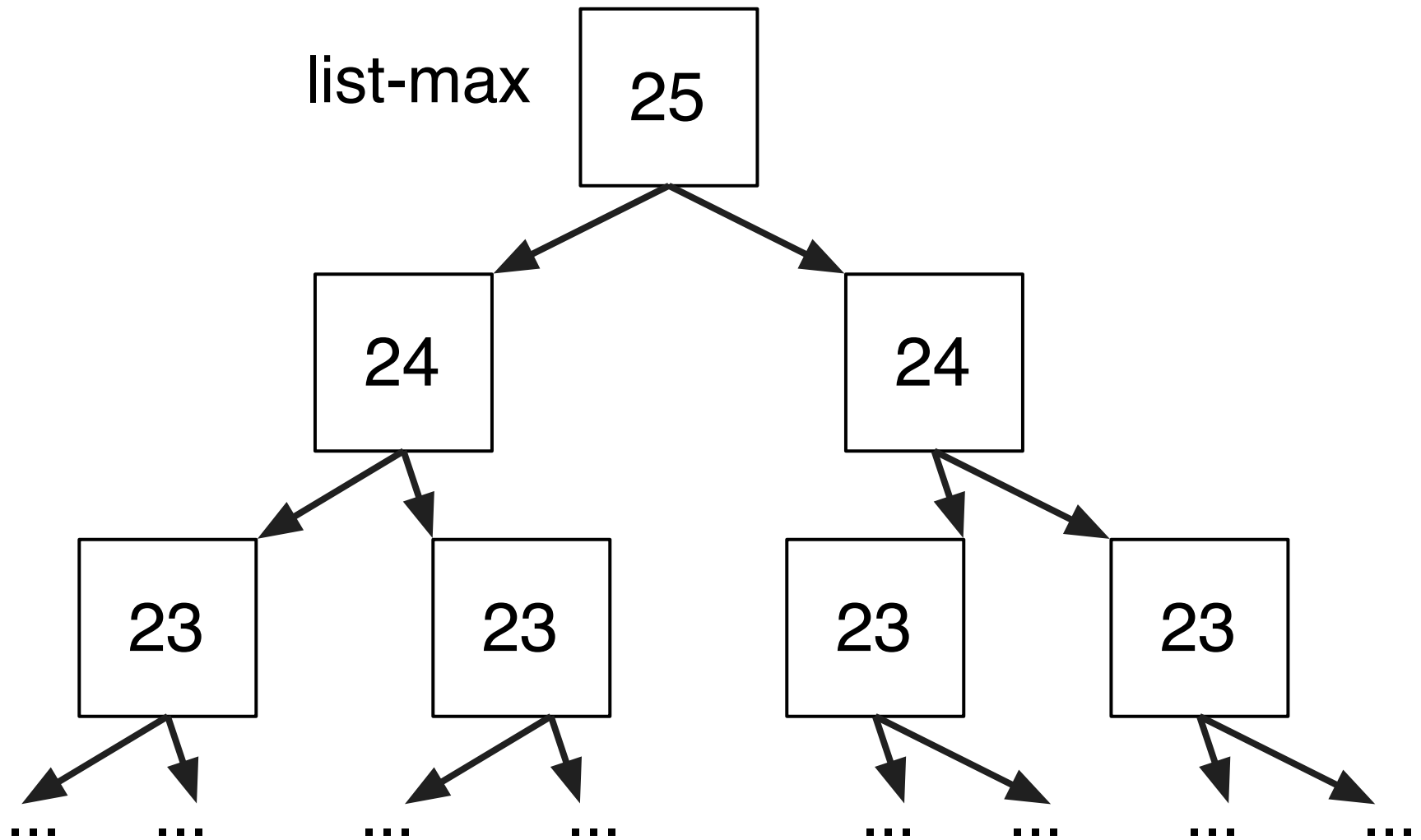
But computing `(max-list (countup-to 1 25))` is very slow.

Why?

The initial application is on a list of length 25.

There are two recursive applications on the rest of this list, which is of length 24.

Each of those makes two recursive applications.



Intuitively, we can find the maximum of a list of numbers by scanning it, remembering the largest value seen so far.

Computationally, we can pass down that largest value seen so far as a parameter called an **accumulator**.

This parameter accumulates the result of prior computation.

This results in the code on the next slide.

:: (max-list/acc lon max-so-far) produces the largest
:: of the maximum element of lon and max-so-far
:: max-list/acc: (listof Num) Num \rightarrow Num

```
(define (max-list/acc lon max-so-far)
  (cond [(empty? lon) max-so-far]
        [(> (first lon) max-so-far)
         (max-list/acc (rest lon) (first lon))]
        [else (max-list/acc (rest lon) max-so-far)]))
```

```
(define (max-list2 lon)
  (max-list/acc (rest lon) (first lon)))
```

Now even `(max-list2 (countup-to 1 2000))` is fast.

```
(max-list2 (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
⇒ (max-list/acc (cons 2 (cons 3 (cons 4 empty))) 1)
⇒ (max-list/acc (cons 3 (cons 4 empty)) 2)
⇒ (max-list/acc (cons 4 empty) 3)
⇒ (max-list/acc empty 4)
⇒ 4
```

This technique is known as **structural recursion with an accumulator**, or sometimes **accumulative recursion**.

It is more difficult to develop and reason about such code, which is why purely structural recursion is preferable if it is appropriate.

HtDP discusses it much later than we are doing (after material we cover in lecture module 10) but in more depth.

Structural vs. accumulative recursion

In **(pure) structural recursion**, all arguments to the recursive function application (or applications, if more than one) are either:

- unchanged, or
- *one step* closer to a base case

In **structural recursion with an accumulator**, arguments are as above, plus one or more accumulators containing partial answers. The accumulatively recursive function is a helper function, and a wrapper function sets the initial value of the accumulator(s).

Another example: reversing a list

`:: my-reverse: (listof Any) → (listof Any)`

`(define (my-reverse lst) ; using pure structural recursion`

`(cond [(empty? lst) empty]`

`[else (append (my-reverse (rest lst)) (list (first lst))))])`

Intuitively, `append` does too much work in repeatedly moving over the produced list to add one element at the end.

This has the same worst-case behaviour as insertion sort.

Reversing a list with an accumulator

```
(define (my-reverse lst) ; wrapper function  
  (my-rev/acc lst empty))
```

```
(define (my-rev/acc lst acc) ; helper function  
  (cond [(empty? lst) acc]  
        [else (my-rev/acc (rest lst)  
                           (cons (first lst) acc))]))
```


A condensed trace

(my-reverse (cons 1 (cons 2 (cons 3 empty))))

⇒ (my-rev/acc (cons 1 (cons 2 (cons 3 empty))) empty)

⇒ (my-rev/acc (cons 2 (cons 3 empty)) (cons 1 empty))

⇒ (my-rev/acc (cons 3 empty) (cons 2 (cons 1 empty)))

⇒ (my-rev/acc empty (cons 3 (cons 2 (cons 1 empty))))

⇒ (cons 3 (cons 2 (cons 1 empty)))

Example: greatest common divisor

In Math 135, you learn that the Euclidean algorithm for GCD can be derived from the following identity for $m > 0$:

$$\gcd(n, m) = \gcd(m, n \bmod m)$$

We also have $\gcd(n, 0) = n$.

We can turn this reasoning directly into a Racket function.

:: (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm

:: euclid-gcd: Nat Nat \rightarrow Nat

```
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

This function does not use structural or accumulative recursion.

The arguments in the recursive application were **generated** by doing a computation on `m` and `n`.

The function `euclid-gcd` uses **generative recursion**.

Once again, functions using generative recursion are easier to get wrong, harder to debug, and harder to reason about.

We will return to generative recursion in a later lecture module.

Structural vs. accumulative vs. generative recursion

In **(pure) structural recursion**, all arguments to the recursive function application (or applications, if there are more than one) are either unchanged, or *one step* closer to a base case.

In **accumulative recursion**, parameters are as above, plus one or more parameters containing partial answers.

In **generative recursion**, parameters are freely calculated at each step. (Watch out for correctness and termination!)

Goals of this module

You should be able to recognize uses of pure structural recursion, accumulative recursion, and generative recursion.

Trees

Readings: HtDP, sections 14, 15, 16.

We will cover the ideas in the text using different examples and different terminology. The readings are still important as an additional source of examples.

Binary arithmetic expressions

A binary arithmetic expression is made up of numbers joined by binary operations $*$, $+$, $/$, and $-$.

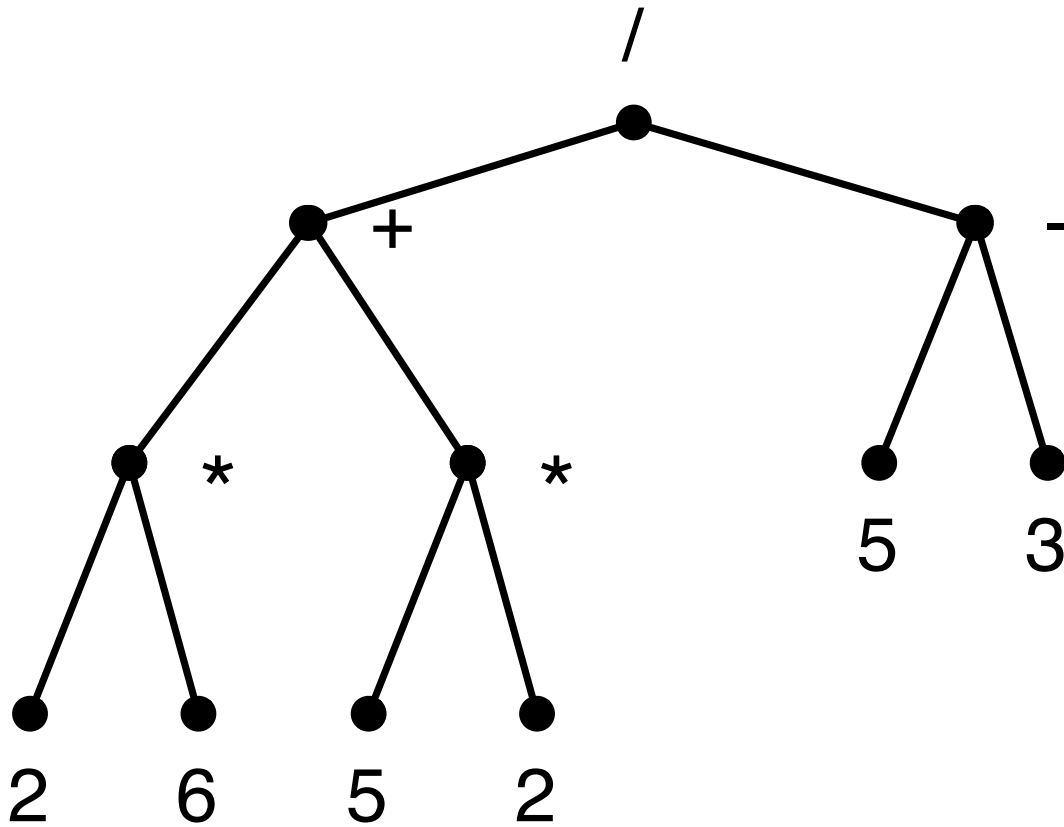
$((2 * 6) + (5 * 2)) / (5 - 3)$ can be defined in terms of *two* smaller binary arithmetic expressions, $(2 * 6) + (5 * 2)$ and $5 - 3$.

Each smaller expression can be defined in terms of even smaller expressions.

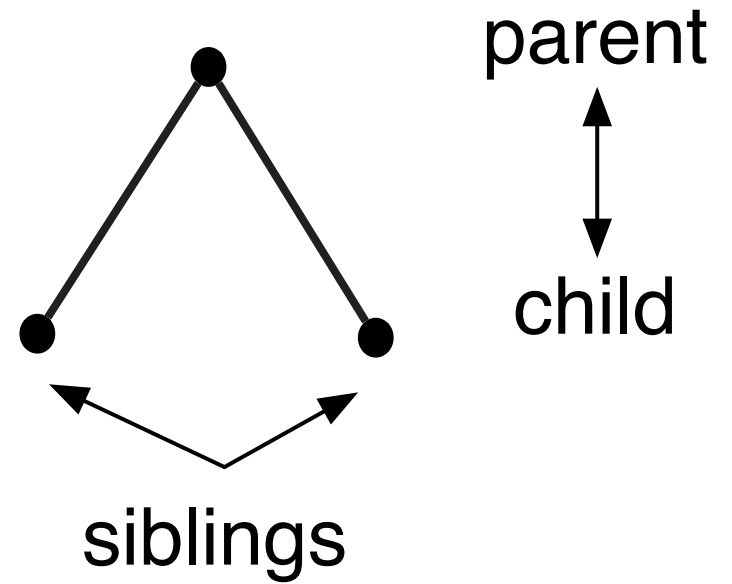
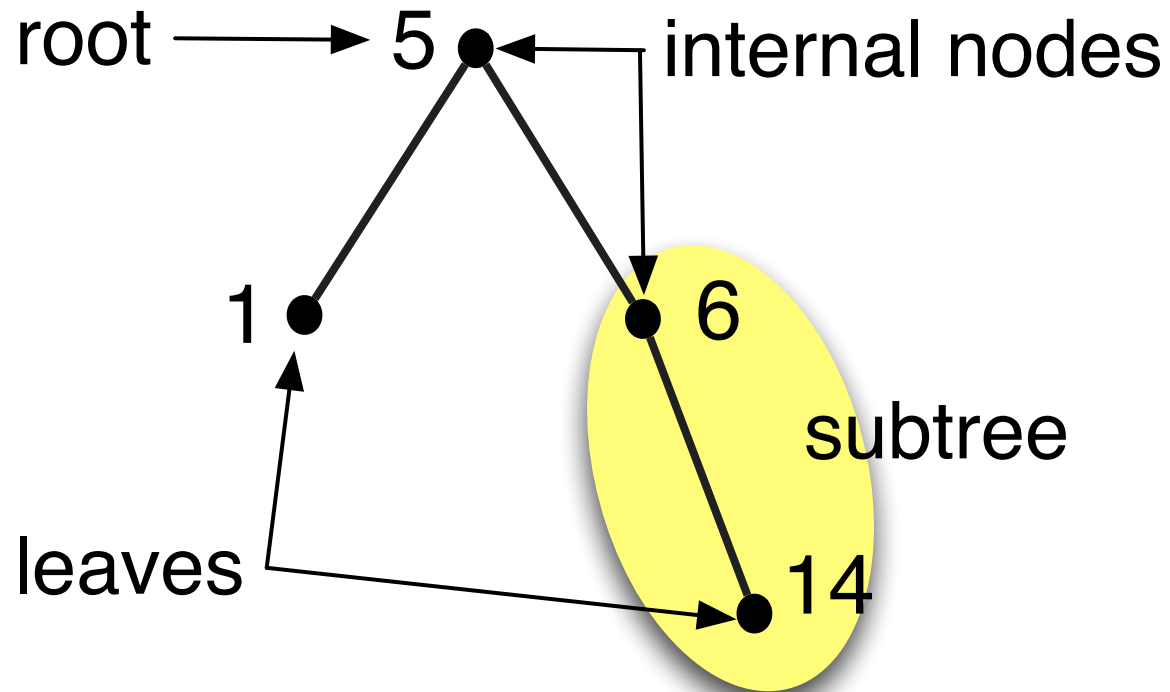
The smallest expressions are numbers.

Visualizing binary arithmetic expressions

$((2 * 6) + (5 * 2)) / (5 - 3)$ can be represented as a **tree**:



Tree terminology



Characteristics of trees

- Number of children of internal nodes:
 - ★ exactly two
 - ★ at most two
 - ★ any number
- Labels:
 - ★ on all nodes
 - ★ just on leaves
- Order of children (matters or not)
- Tree structure (from data or for convenience)

Representing binary arithmetic expressions

Internal nodes each have exactly two children.

Leaves have number labels.

Internal nodes have symbol labels.

We care about the order of children.

The structure of the tree is dictated by the expression.

How can we group together information for an internal node?

How can we allow different definitions for leaves and internal nodes?

(define-struct binode (op arg1 arg2))

:: A Binary arithmetic expression Internal Node (BINode)

:: is a (make-binode (anyof '* '+ '/ '—) BinExp BinExp)

:: A binary arithmetic expression (BinExp) is one of:

:: ★ a Num

:: ★ a BINode

:: Examples:

5

(make-binode '* 2 6)

(make-binode '+ 2 (make-binode '— 5 3))

A more complex example:

```
(make-binode '/  
  (make-binode '+ (make-binode '* 2 6)  
    (make-binode '* 5 2))  
  (make-binode '- 5 3))
```

Template for binary arithmetic expressions

The only new idea in forming the template is the application of the recursive function to *each* piece that satisfies the data definition.

```
:: my-binexp-fn: BinExp  $\rightarrow$  Any
```

```
(define (my-binexp-fn ex)
```

```
  (cond [(number? ex) ...]
```

```
        [else (... (binode-op ex) ...
```

```
                  (my-binexp-fn (binode-arg1 ex)) ...
```

```
                  (my-binexp-fn (binode-arg2 ex)) ... )]))
```

Evaluation of expressions

```
(define (eval ex)
  (cond [(number? ex) ex]
        [(symbol=? (binode-op ex) '*)
         (* (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))]
        [(symbol=? (binode-op ex) '+)
         (+ (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))]
        [(symbol=? (binode-op ex) '/')
         (/ (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))]
        [(symbol=? (binode-op ex) '—)
         (— (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))]))
```



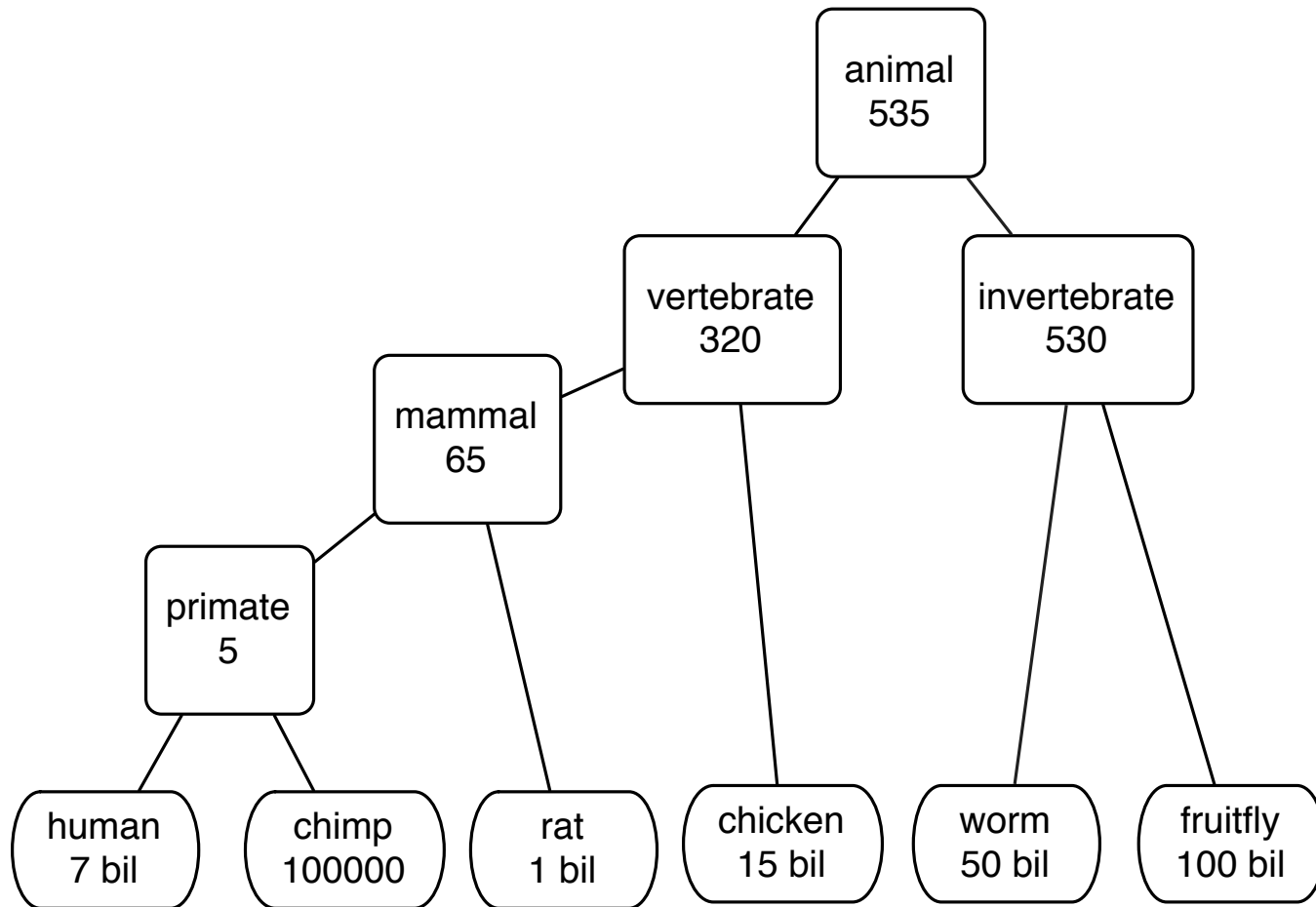
```
(define (eval ex)
  (cond [(number? ex) ex]
        [else (eval-binode (binode-op ex)
                             (eval (binode-arg1 ex))
                             (eval (binode-arg2 ex))))]))
```

```
(define (eval-binode op left right)
  (cond [(symbol=? op '—) (— left right)]
        [(symbol=? op '+) (+ left right)]
        [(symbol=? op '/') (/ left right)]
        [(symbol=? op '*') (* left right)]
        [else (error "Unrecognized operator")]))
```

Evolution trees

- a data structure recording information about the evolution of species
- contains two kinds of information:
 - modern species (e.g. humans) with population count
 - evolutionary history via **evolution events**, with an estimate of how long ago the event occurred

An “evolution event” leads to a splitting of one species into two distinct species (for example, through physical separation).



Representing evolution trees

Internal nodes each have exactly two children.

Leaves have names and populations of modern species.

Internal nodes have names and dates of evolution events.

The order of children does not matter.

The structure of the tree is dictated by a hypothesis about evolution.

Data definitions for evolution trees

;; A Taxon is one of:

;; ★ a Modern

;; ★ an Ancient

(**define-struct** modern (name pop))

;; A Modern is a (make-modern Str Nat)

(**define-struct** ancient (name age left right))

;; An Ancient is a (make-ancient Str Num Taxon Taxon)

Note that the Ancient data definition uses a pair of Taxons.

Here are the definitions for our example tree drawing.

```
(define-struct modern (name pop))
```

```
(define-struct ancient (name age left right))
```

```
(define human (make-modern "human" 6.8e9))
```

```
(define chimp (make-modern "chimpanzee" 1.0e5))
```

```
(define rat (make-modern "rat" 1.0e9))
```

```
(define chicken (make-modern "chicken" 1.5e10))
```

```
(define worm (make-modern "worm" 5.0e10))
```

```
(define fruit-fly (make-modern "fruit fly" 1.0e11))
```

```
(define primate (make-ancient "Primate" 5 human chimp))  
(define mammal (make-ancient "Mammal" 65 primate rat))  
(define vertebrate  
  (make-ancient "Vertebrate" 320 mammal chicken))  
(define invertebrate  
  (make-ancient "Invertebrate" 530 worm fruit-fly))  
(define animal (make-ancient "Animal" 535 vertebrate invertebrate))
```

Derive the template for taxon computations from the data definition.

```
:: my-taxon-fn: Taxon → Any
```

```
(define (my-taxon-fn t)  
  (cond [(modern? t) (my-modern-fn t)]  
        [(ancient? t) (my-ancient-fn t)]))
```

This is a straightforward implementation based on the data definition. It's also a good strategy to take a complicated problem (dealing with a Taxon) and decompose it into simpler problems (dealing with a Modern or an Ancient).

Functions for these two data definitions are on the next slide.

:: my-modern-fn: Modern \rightarrow Any

```
(define (my-modern-fn t)  
  (... (modern-name t) ...  
        (modern-pop t) ... ))
```

:: my-ancient-fn: Ancient \rightarrow Any

```
(define (my-ancient-fn t)  
  (... (ancient-name t) ...  
        (ancient-age t) ...  
        (ancient-left t) ...  
        (ancient-right t) ... ))
```

We know that `(ancient-left t)` and `(ancient-right t)` are Taxons, so apply the Taxon-processing function to them.

:: my-ancient-fn: Ancient \rightarrow Any

```
(define (my-ancient-fn t)
  (... (ancient-name t) ...
        (ancient-age t) ...
        (my-taxon-fn (ancient-left t)) ...
        (my-taxon-fn (ancient-right t)) ... ))
```

`my-ancient-fn` uses `my-taxon-fn` and `my-taxon-fn` uses `my-ancient-fn`. This is called *mutual recursion*.

A function on taxons

This function counts the number of modern descendant species of a taxon. A taxon is its own descendant.

`:: ndesc-species: Taxon \rightarrow Nat`

```
(define (ndesc-species t)
  (cond [(modern? t) (ndesc-modern t)]
        [(ancient? t) (ndesc-ancient t)]))
```

```
(check-expect (ndesc-species animal) 6)
```

```
(check-expect (ndesc-species human) 1)
```

:: ndesc-modern: Modern \rightarrow Nat

```
(define (ndesc-modern t)  
  1)
```

:: ndesc-ancient: Ancient \rightarrow Nat

```
(define (ndesc-ancient t)  
  (+ (ndesc-species (ancient-left t))  
     (ndesc-species (ancient-right t))))
```

Counting evolution events

:: (recent-events t n) produces the number of evolution events in t

:: taking place within the last n million years

:: recent-events: Taxon Num \rightarrow Nat

:: Examples:

(check-expect (recent-events worm 500) 0)

(check-expect (recent-events animal 500) 3)

(check-expect (recent-events animal 530) 4)

(define (recent-events t n) ...)

For a more complicated computation, try to compute the list of species that connects two specified taxons, if such a list exists.

The function `ancestors` consumes a taxon `from` and a modern species `to`. If `from` is the ancestor of `to`, `ancestors` produces the list of names of the species that connect `from` to `to` (including the names of `from` and `to`). Otherwise, it produces `false`.

What cases should the examples cover?

- The taxon `from` can be either a `modern` or a `ancient`.
- The function can produce either `false` or a list of strings.

:: (ancestors from to) produces chain of names from...to

:: ancestors: Taxon Modern \rightarrow (anyof (listof Str) false)

:: Examples

(check-expect (ancestors human human) '("human"))

(check-expect (ancestors mammal worm) false)

(check-expect (ancestors mammal human)

'("Mammal" "Primate" "human"))

(define (ancestors from to)

(cond [(modern? from) (ancestors-modern from to)]

[(ancient? from) (ancestors-ancient from to)]))

```
(define (ancestors-modern from to)
  (cond [(equal? from to) (list (modern-name to))]
        [else false]))
```

```
(define (ancestors-ancient from to)
  (... (ancient-name from) ...
        (ancient-age from) ...
        (ancestors (ancient-left from) to) ...
        (ancestors (ancient-right from) to) ... ))
```



```
::; ancestors-ancient: Ancient Modern → (anyof (listof Str) false)
(check-expect (ancestors-ancient mammal worm) false)
(check-expect (ancestors-ancient mammal rat) '("Mammal" "rat"))
(check-expect (ancestors-ancient mammal human)
              '("Mammal" "Primate" "human"))
(define (ancestors-ancient from to)
  (cond
    [(cons? (ancestors (ancient-left from) to))
     (cons (ancient-name from) (ancestors (ancient-left from) to))]
    [(cons? (ancestors (ancient-right from) to))
     (cons (ancient-name from) (ancestors (ancient-right from) to))]
    [else false]))
```

When we try filling in the recursive case, we notice that we have to use the results of the recursive function application more than once.

To avoid repeating the computation, we can pass the results of the recursive function applications (plus whatever other information is needed) to a helper function.

A more efficient ancestors-ancient function

```
(define (ancestors-ancient from to)
  (extend-list (ancient-name from)
               (ancestors (ancient-left from) to)
               (ancestors (ancient-right from) to)))
```

```
(define (extend-list from-n from-l from-r)
  (cond [(cons? from-l) (cons from-n from-l)]
        [(cons? from-r) (cons from-n from-r)]
        [else false]))
```

Binary trees

Evolution trees are an example of **binary trees**: trees with at most two children for each node.

Binary trees are a fundamental part of computer science, independently of what language you use.

Next we will use another type of binary tree to provide a more efficient implementation of dictionaries.

Dictionaries revisited

Recall from module 06 that a dictionary stores a set of (key, value) pairs, with at most one occurrence of any key.

It supports lookup, add, and remove operations.

We implemented a dictionary as an association list of two-element lists.

This implementation had the problem that a search could require looking through the entire list, which will be inefficient for large dictionaries.

Our new implementation will put the (key,value) pairs into a **binary tree** instead of a **list** in order to quickly search large dictionaries.
(We'll see how soon.)

```
(define-struct node (key val left right))
```

```
:: A Node is a (make-node Num Str BT BT)
```

```
:: A binary tree (BT) is one of:
```

```
:: ★ empty
```

```
:: ★ Node
```

What is the template?

Counting values in a Binary Tree

Let us fill in the template to make a simple function: count how many nodes in the BT have a value equal to v :

$:: \text{count-values: BT Str} \rightarrow \text{Nat}$

```
(define (count-values tree v)
  (cond [(empty? tree) 0]
        [else (+ (cond [(string=? v (node-val tree)) 1]
                        [else 0])
                  (count-values (node-left tree) v)
                  (count-values (node-right tree) v))]))
```

Add 1 to every key in a given tree:

:: increment: BT \rightarrow BT

```
(define (increment tree)
  (cond [(empty? tree) empty]
        [else (make-node (add1 (node-key tree))
                           (node-val tree)
                           (increment (node-left tree))
                           (increment (node-right tree))))]))
```


Searching binary trees

We are now ready to try to search our binary tree to produce the value associated with the given key (or **false** if the key is not in the dictionary).

Our strategy:

- See if the root node contains the key we're looking for. If so, produce the associated value.
- Otherwise, recursively search in the left subtree, and in the right subtree. If either recursive search comes up with a string, produce that string. Otherwise, produce **false**.

Much as with [ancestors-ancient](#), where we were searching an evolution tree, we will find a helper function to be useful:

```
:: (pick-string a b) produces a if it is a string,  
::   or b if it is a string, otherwise false  
;; pick-string: (anyof Str false) (anyof Str false) → (anyof Str false)  
(check-expect (pick-string false "hi") "hi")  
(check-expect (pick-string "there" false) "there")  
(check-expect (pick-string false false) false)  
(define (pick-string a b)  
  (cond [(string? a) a]  
        [(string? b) b]  
        [else false]))
```

Now we can fill in our BT template to write our search function:

```
:: search-bt: Num BT → (anyof Str false)

(define (search-bt n tree)
  (cond [(empty? tree) false]
        [(= n (node-key tree)) (node-val tree)]
        [else (pick-string
                  (search-bt n (node-left tree))
                  (search-bt n (node-right tree))))]))
```

Is this more efficient than searching an AL?

Binary search trees

We will now make one change that will make searching **much** more efficient. This change will create a tree structure known as a **binary search tree** (BST).

The (key,value) pairs are stored at the nodes of a tree.

For any given collection of keys and values, there is more than one possible tree.

The placement of pairs in nodes can make it possible to improve the running time compared to association lists.

:: A Binary Search Tree (BST) is one of:

:: ★ empty

:: ★ a Node

(define-struct node (key val left right))

:: A Node is a (make-node Num Str BST BST)

:: requires: key > every key in left BST

:: key < every key in right BST

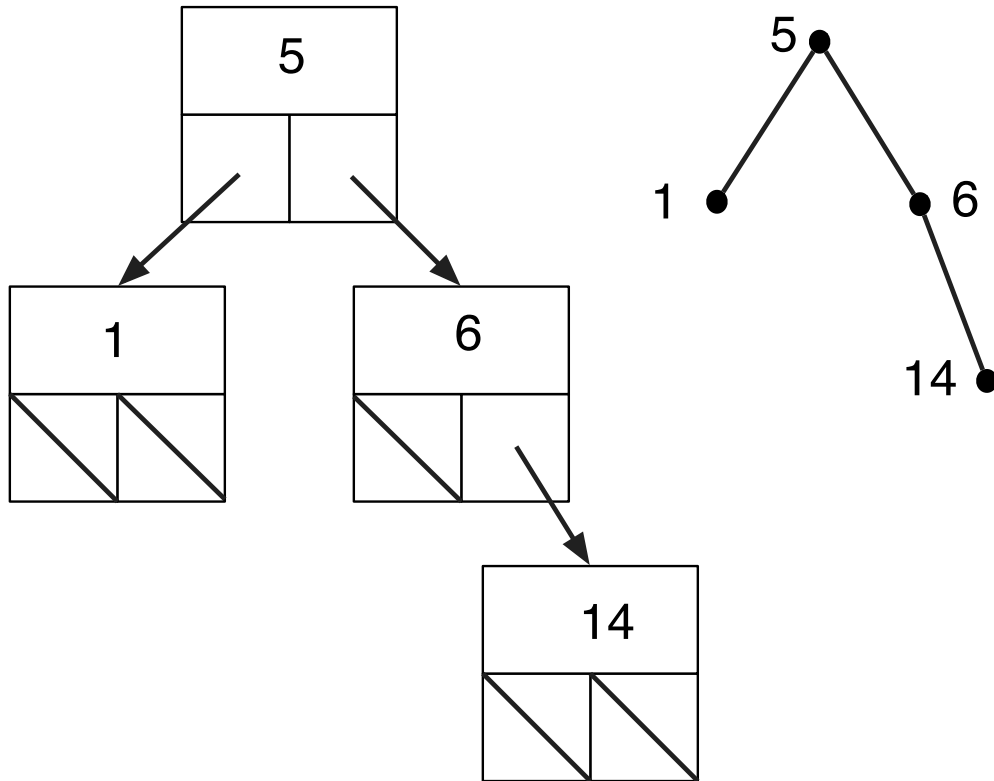
The BST ordering property:

- key is greater than every key in left.
- key is less than every key in right.

A BST example

```
(make-node 5 "John"  
  (make-node 1 "Alonzo" empty empty)  
  (make-node 6 "Alan"  
    empty  
    (make-node 14 "Ada"  
      empty  
      empty))))
```

Drawing BSTs



(Note: the value field is not represented.)

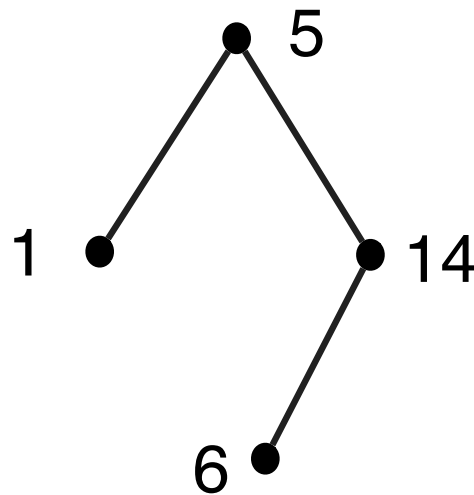
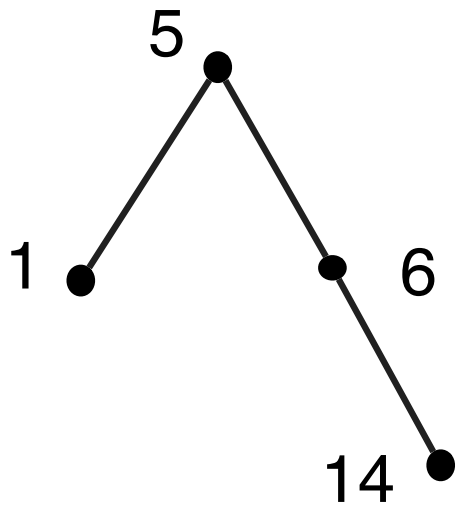
We have made several minor changes from HtDP:

- We use `empty` instead of `false` for the base case.
- We use `key` instead of `ssn`, and `val` instead of `name`.

The value can be any Racket value.

We can generalize to other key types with an ordering (e.g. strings, using `string<?`).

There can be several BSTs holding a particular set of (key, value) pairs.



Making use of the ordering property

Main advantage: for certain computations, one of the recursive function applications in the template can always be avoided.

This is more efficient (sometimes considerably so).

In the following slides, we will demonstrate this advantage for searching and adding.

We will write the code for searching, and briefly sketch adding, leaving you to write the Racket code.

Searching in a BST

How do we search for a key n in a BST?

We reason using the data definition of **BST**.

If the BST is **empty**, then n is not in the BST.

If the BST is of the form **(make-node k v l r)**, and k equals n , then we have found it.

Otherwise it might be in either of the trees l , r .

If $n < k$, then n cannot be in r , and we only need to recursively search in l .

If $n > k$, then n cannot be in l , and we only need to recursively search in r .

Either way, we save one recursive function application.

:: (search-bst n t) produces the value associated with n,

:: or false if n is not in t

:: search-bst: Num BST \rightarrow (anyof Str false)

(define (search-bst n t)

(cond [(empty? t) false]

[(= n (node-key t)) (node-val t)]

[(< n (node-key t)) (search-bst n (node-left t))]

[(> n (node-key t)) (search-bst n (node-right t))]))

Creating a BST

How do we create a BST from a list of (key, value) pairs?

We reason using the data definition of a list.

If the list is empty, the BST is **empty**.

If the list is of the form **(cons (list k v) lst)**, we add the pair **(k, v)** to the BST created from the list **lst**.

Adding to a BST

How do we add a pair (k, v) to a BST `bstree`?

If `bstree` is `empty`, then the result is a BST with only one node.

Otherwise `bstree` is of the form `(make-node n w l r)`.

If $k = n$, we form the tree with k , v , l , and r (we replace the old value by v).

If $k < n$, then the pair must be added to l , and if $k > n$, then the pair must be added to r . Again, we need only make one recursive function application.

Binary search trees in practice

If the BST has all left subtrees empty, it looks and behaves like a sorted association list, and the advantage is lost.

In later courses, you will see ways to keep a BST “balanced” so that “most” nodes have nonempty left and right children.

By that time you will better understand how to analyze the efficiency of algorithms and operations on data structures.

General trees

Binary trees can be used for a large variety of application areas.

One limitation is the restriction on the number of children.

How might we represent a node that can have up to three children?

What if there can be any number of children?

General arithmetic expressions

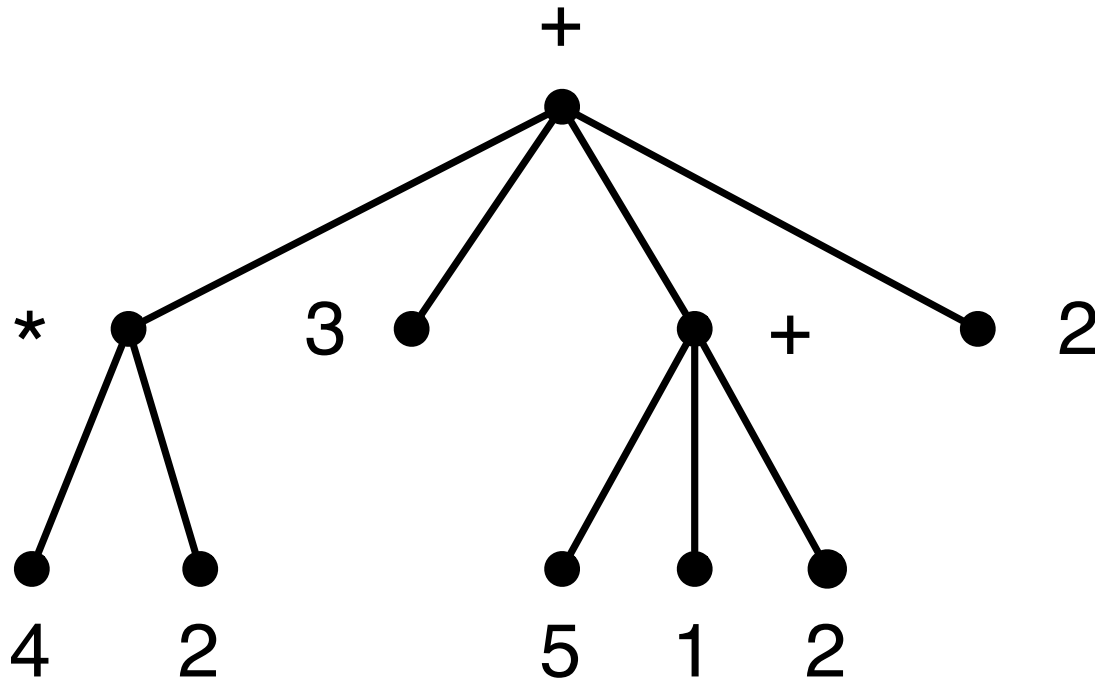
For binary arithmetic expressions, we formed binary trees.

Racket expressions using the functions $+$ and $*$ can have an unbounded number of arguments.

For simplicity, we will restrict the operations to $+$ and $*$.

$(+ (* 4 2) 3 (+ 5 1 2) 2)$

We can visualize an arithmetic expression as a general tree.



$(+ (* 4 2) 3 (+ 5 1 2) 2)$

For a binary arithmetic expression, we defined a structure with three fields: the operation, the first argument, and the second argument.

For a general arithmetic expression, we define a structure with two fields: the operation and a list of arguments (which is a list of arithmetic expressions).

We also need the data definition of a list of arithmetic expressions.

(define-struct ainode (op args))

:: a Arithmetic expression Internal Node (AINode)

:: is a (make-ainode (anyof '* '+') (listof AExp))

:: An Arithmetic Expression (AExp) is one of:

:: ★ a Num

:: ★ an AINode

Each definition depends on the other, and each template will depend on the other. We saw this in the evolution trees example, as well. A Taxon was defined in terms of a modern species and an ancient species. The ancient species was defined in terms of a Taxon.

Examples of arithmetic expressions:

3

(make-ainode '+ (list 3 4))

(make-ainode '* (list 3 4))

(make-ainode '+ (list (make-ainode '* '(4 2)) 3
 (make-ainode '+ '(5 1 2)) 2))

It is also possible to have an operation and an empty list; recall substitution rules for **and** and **or**.

Templates for arithmetic expressions

```
(define (my-aexp-fn ex)
  (cond [(number? ex) ...]
        [else (... (ainode-op ex) ...
                    (my-listof-aexp-fn (ainode-args ex)) ... )]))
```

```
(define (my-listof-aexp-fn exlist)
  (cond [(empty? exlist) ...]
        [else (... (my-aexp-fn (first exlist)) ...
                    (my-listof-aexp-fn (rest exlist)) ... )]))
```

The function eval

:: eval: AExp \rightarrow Num

```
(define (eval ex)
  (cond [(number? ex) ex]
        [else (apply (ainode-op ex) (ainode-args ex))]))
```


:: apply: Sym (listof AExp) \rightarrow Num

(define (apply f exlist)

(cond [(empty? exlist) (cond [(symbol=? f '*) 1]
[(symbol=? f '+) 0])]

[(symbol=? f '*)

(* (eval (first exlist)) (apply f (rest exlist))))]

[(symbol=? f '+)

(+ (eval (first exlist)) (apply f (rest exlist))))]]))

A simplified apply

:: apply: Sym (listof AExp) \rightarrow Num

```
(define (apply f exlist)
  (cond [(and (empty? exlist) (symbol=? f '*)) 1]
        [(and (empty? exlist) (symbol=? f '+)) 0]
        [(symbol=? f '*)
         (* (eval (first exlist)) (apply f (rest exlist)))]
        [(symbol=? f '+)
         (+ (eval (first exlist)) (apply f (rest exlist)))]))
```

Condensed trace of aexp evaluation

```
(eval (make-ainode '+ (list (make-ainode '* '(3 4))  
                             (make-ainode '* '(2 5)))))
```

```
⇒ (apply '+ (list (make-ainode '* '(3 4))  
                  (make-ainode '* '(2 5))))
```

```
⇒ (+ (eval (make-ainode '* '(3 4)))  
     (apply '+ (list (make-ainode '* '(2 5)))))
```

```
⇒ (+ (apply '* '(3 4))  
     (apply '+ (list (make-ainode '* '(2 5)))))
```

$\Rightarrow (+ (* (\text{eval } 3) (\text{apply } ' * '(4))))$
 $(\text{apply } ' + (\text{list } (\text{make-ainode } ' * '(2\ 5))))$
 $\Rightarrow (+ (* 3 (\text{apply } ' * '(4))))$
 $(\text{apply } ' + (\text{list } (\text{make-ainode } ' * '(2\ 5))))$
 $\Rightarrow (+ (* 3 (* (\text{eval } 4) (\text{apply } ' * \text{empty}))))$
 $(\text{apply } ' + (\text{list } (\text{make-ainode } ' * '(2\ 5))))$
 $\Rightarrow (+ (* 3 (* 4 (\text{apply } ' * \text{empty}))))$
 $(\text{apply } ' + (\text{list } (\text{make-ainode } ' * '(2\ 5))))$

```

⇒ (+ (* 3 (* 4 1))
      (apply '+ (list (make-ainode '* '(2 5)))))
⇒ (+ 12
      (apply '+ (list (make-ainode '* '(2 5)))))
⇒ (+ 12 (+ (eval (make-ainode '* '(2 5)))
            (apply '+ empty)))
⇒ (+ 12 (+ (apply '* '(2 5))
            (apply '+ empty)))
⇒ (+ 12 (+ (* (eval 2) (apply '* '(5)))
            (apply '+ empty)))

```

$\Rightarrow (+\ 12\ (+\ (*\ 2\ (\text{apply}\ '*\ '(5)))$
 $\qquad\qquad\qquad (\text{apply}\ '+\ \text{empty})))$

$\Rightarrow (+\ 12\ (+\ (*\ 2\ (*\ (\text{eval}\ 5)\ (\text{apply}\ '*\ \text{empty})))$
 $\qquad\qquad\qquad (\text{apply}\ '+\ \text{empty})))$

$\Rightarrow (+\ 12\ (+\ (*\ 2\ (*\ 5\ (\text{apply}\ '*\ \text{empty})))$
 $\qquad\qquad\qquad (\text{apply}\ '+\ \text{empty})))$

$\Rightarrow (+\ 12\ (+\ (*\ 2\ (*\ 5\ 1))$
 $\qquad\qquad\qquad (\text{apply}\ '+\ \text{empty})))$

$\Rightarrow (+\ 12\ (+\ (*\ 2\ 5)\ (\text{apply}\ '+\ \text{empty})))$

$\Rightarrow (+\ 12\ (+\ 10\ (\text{apply}\ '+\ \text{empty})))$

$\Rightarrow (+\ 12\ (+\ 10\ 0)) \Rightarrow (+\ 12\ 10) \Rightarrow 22$

Alternate data definition

In Module 6, we saw how a list could be used instead of a structure holding student information.

Here we could use a similar idea to replace the structure `ainode` and the data definitions for `AExp`.

:: An alternate arithmetic expression (AltAExp) is one of:

:: ★ a Num

:: ★ (cons (anyof '* '+) (listof AltAExp))

Each expression is a list consisting of a symbol (the operation) and a list of expressions.

3

'(+ 3 4)

'(+ (* 4 2 3) (+ (* 5 1 2) 2))

Templates: AltAExp and (listof AltAExp)

```
(define (my-altaexp-fn ex)
  (cond [(number? ex) ...]
        [else (... (first ex) ...
                    (my-listof-altaexp-fn (rest ex)) ... )]))
```

```
(define (my-listof-altaexp-fn exlist)
  (cond [(empty? exlist) ...]
        [(cons? exlist) (... (my-altaexp-fn (first exlist)) ...
                              (my-listof-altaexp-fn (rest exlist)) ... )]))
```

:: eval: AltAExp \rightarrow Num

```
(define (eval aax)
  (cond [(number? aax) aax]
        [else (apply (first aax) (rest aax))]))
```

:: apply: Sym AltAExpList \rightarrow Num

```
(define (apply f aaxl)
  (cond [(and (empty? aaxl) (symbol=? f '*)) 1]
        [(and (empty? aaxl) (symbol=? f '+)) 0]
        [(symbol=? f '*)
         (* (eval (first aaxl)) (apply f (rest aaxl)))]
        [(symbol=? f '+)
         (+ (eval (first aaxl)) (apply f (rest aaxl)))]))
```

A condensed trace

`(eval '(* (+ 1 2) 4))`

\Rightarrow `(apply '* '((+ 1 2) 4))`

\Rightarrow `(* (eval '(+ 1 2)) (apply '* '(4)))`

\Rightarrow `(* (apply '+ '(1 2)) (apply '* '(4)))`

\Rightarrow `(* (+ 1 (apply '+ '(2))) (apply '* '(4)))`

\Rightarrow `(* (+ 1 (+ 2 (apply '+ '()))) (apply '* '(4)))`

\Rightarrow `(* (+ 1 (+ 2 0)) (apply '* '(4)))`

$\Rightarrow (* (+ 1 2) (\text{apply } ' * '(4)))$

$\Rightarrow (* 3 (\text{apply } ' * '(4)))$

$\Rightarrow (* 3 (* 4 (\text{apply } ' * '())))$

$\Rightarrow (* 3 (* 4 1))$

$\Rightarrow (* 3 4)$

$\Rightarrow 12$

Structuring data using mutual recursion

Mutual recursion arises when complex relationships among data result in cross references between data definitions.

The number of data definitions can be greater than two.

Structures and lists may also be used.

In each case:

- create templates from the data definitions and
- create one function for each template.

Other uses of general trees

We can generalize from allowing only two arithmetic operations and numbers to allowing arbitrary functions and variables.

In effect, we have the beginnings of a Racket interpreter.

But beyond this, the type of processing we have done on arithmetic expressions can be applied to tagged hierarchical data, of which a Racket expression is just one example.

Organized text and Web pages provide other examples.

```
'(chapter
  (section
    (paragraph "This is the first sentence."
      "This is the second sentence.")
    (paragraph "We can continue in this manner."))
  (section ...)
  ...
)
```



```
'(webpage
  (title "CS 115: Introduction to Computer Science 1 ")
  (paragraph "For a course description, "
    (link "click here." "desc.html")
    "Enjoy the course!")
  (horizontal-line)
  (paragraph "(Last modified yesterday.)" ))
```

Nested lists

So far, we have discussed flat lists (no nesting):

```
(list 'a 1 "hello" 'x)
```

and lists of lists (one level of nesting):

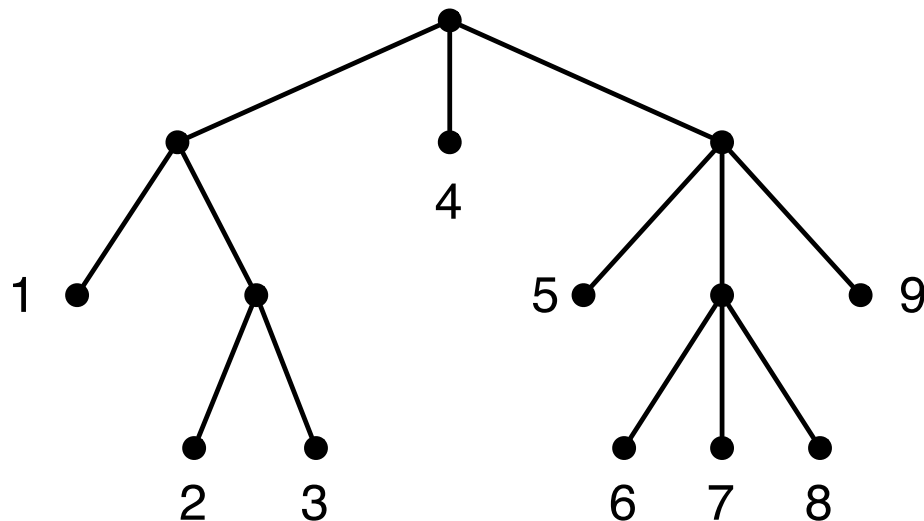
```
(list (list 1 "a") (list 2 "b"))
```

We now consider **nested lists** (arbitrary nesting):

```
'((1 (2 3)) 4 (5 (6 7 8) 9))
```

Nested lists as leaf-labelled trees

It is often useful to visualize a nested list as a **leaf-labelled tree**, in which the leaves correspond to the elements of the list, and the internal nodes indicate the nesting:



'((1 (2 3)) 4 (5 (6 7 8) 9))'

Examples of nested lists:

empty

'(4 2)

'((4 2) 3 (4 1 6))

'((3) 2 () (4 (3 6)))

Each nonempty tree is a list of subtrees.

The first subtree in the list is either

- a single leaf (not a list) or
- a subtree rooted at an internal node (a list).

Data definition for nested lists

:: A nested list of numbers (Nest-List-Num) is one of:

:: ★ `empty`

:: ★ `(cons Num Nest-List-Num)`

:: ★ `(cons Nest-List-Num Nest-List-Num)`

Generic data definition for nested lists

:: A nested list of X (Nest-List-X) is one of:

:: ★ **empty**

:: ★ (**cons** X Nest-List-X)

:: ★ (**cons** Nest-List-X Nest-List-X)

Template for nested lists

The template follows from the data definition.

```
(define (my-nest-lst-fn lst)
  (cond [(empty? lst) ...]
        [(X? (first lst))
         (... (first lst) ... (my-nest-lst-fn (rest lst)) ...)]
        [else
         (... (my-nest-lst-fn (first lst)) ...
              (my-nest-lst-fn (rest lst)) ... )]))
```

The function count-items

:: count-items: Nest-List-Num \rightarrow Nat

```
(define (count-items nln)
  (cond [(empty? nln) 0]
        [(number? (first nln))
         (+ 1 (count-items (rest nln)))]
        [else (+ (count-items (first nln))
                  (count-items (rest nln)))]))
```


Condensed trace of count-items

`(count-items '((10 20) 30))`

$\Rightarrow (+ (\text{count-items } '(10\ 20)) (\text{count-items } '(30)))$

$\Rightarrow (+ (+\ 1 (\text{count-items } '(20))) (\text{count-items } '(30)))$

$\Rightarrow (+ (+\ 1 (+\ 1 (\text{count-items } '()))) (\text{count-items } '(30)))$

$\Rightarrow (+ (+\ 1 (+\ 1\ 0)) (\text{count-items } '(30)))$

$\Rightarrow (+ (+\ 1\ 1) (\text{count-items } '(30)))$

$\Rightarrow (+\ 2 (\text{count-items } '(30)))$

$\Rightarrow (+\ 2 (+\ 1 (\text{count-items } '())))$

$\Rightarrow (+\ 2 (+\ 1\ 0)) \Rightarrow (+\ 2\ 1) \Rightarrow 3$

Flattening a nested list

`flatten` produces a flat list from a nested list.

```
;; flatten: Nest-List-Num → (listof Num)
```

```
(define (flatten lst) ... )
```

We make use of the built-in Racket function `append`.

```
(append '(1 2) '(3 4)) ⇒ '(1 2 3 4)
```

Remember: use `append` only when the first list has length greater than one, or there are more than two lists.

;; flatten: Nest-List-Num \rightarrow (listof Num)

```
(define (flatten lst)
  (cond [(empty? lst) empty]
        [(number? (first lst))
         (cons (first lst) (flatten (rest lst)))]
        [else (append (flatten (first lst))
                        (flatten (rest lst)))])])
```

Condensed trace of flatten

(flatten '((10 20) 30))

⇒ (append (flatten '(10 20)) (flatten '(30)))

⇒ (append (cons 10 (flatten '(20))) (flatten '(30)))

⇒ (append (cons 10 (cons 20 (flatten '()))) (flatten '(30)))

⇒ (append (cons 10 (cons 20 empty)) (flatten '(30)))

⇒ (append (cons 10 (cons 20 empty)) (cons 30 (flatten '())))

⇒ (append (cons 10 (cons 20 empty)) (cons 30 empty))

⇒ (cons 10 (cons 20 (cons 30 empty)))

Goals of this module

You should be familiar with tree terminology.

You should understand the data definitions for binary arithmetic expressions, evolution trees, and binary search trees, understand how the templates are derived from those definitions, and how to use the templates to write functions that consume those types of data.

You should understand the definition of a binary search tree and its ordering property.

You should be able to write functions which consume binary search trees, including those sketched (but not developed fully) in lecture.

You should be able to develop and use templates for other binary trees, not necessarily presented in lecture.

You should understand the idea of mutual recursion for both examples given in lecture and new ones that might be introduced in lab, assignments, or exams.

You should be able to develop templates from mutually recursive data definitions, and to write functions using the templates.

Local definitions and lexical scope

Readings: HtDP, Intermezzo 3 (Section 18).

Language level: Intermediate Student

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.

```
(local [(define x1 exp1) ... (define xn expn)] bodyexp)
```

What use is this?

Motivating local definitions

Consider Heron's formula for the area of a triangle with sides a , b , c : $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a + b + c)/2$.

It is not hard to create a Racket function to compute this function, but it is difficult to do so in a clear and natural fashion.

We will describe several possibilities, starting with a direct implementation.

```
(define (t-area a b c)
  (sqrt
    (* (/ (+ a b c) 2)
      (− (/ (+ a b c) 2) a)
      (− (/ (+ a b c) 2) b)
      (− (/ (+ a b c) 2) c))))
```

The repeated computation of $s = (a + b + c)/2$ is awkward.

We could notice that $s - a = (-a + b + c)/2$, and make similar substitutions.

```
(define (t-area a b c)
  (sqrt
    (* (/ (+ a b c) 2)
      (/ (+ (- a) b c) 2)
      (/ (+ a (- b) c) 2)
      (/ (+ a b (- c)) 2))))
```

This is short, but its relationship to Heron's formula is unclear from just reading the code, and the technique does not generalize.

We could instead use a helper function.

```
(define (t-area2 a b c)
  (sqrt
    (* (s a b c)
      (— (s a b c) a)
      (— (s a b c) b)
      (— (s a b c) c)))))
```

```
(define (s a b c)
  (/ (+ a b c) 2))
```

This generalizes well to formulas that define several intermediate quantities.

But the helper functions need parameters, which again makes the relationship to Heron's formula hard to see.

We could instead move the computation with a known value of s into a helper function, and provide the value of s as a parameter.

```
(define (t-area3/s a b c s)
  (sqrt (* s (- s a) (- s b) (- s c))))
(define (t-area3 a b c)
  (t-area3/s a b c (/ (+ a b c) 2)))
```

This is more readable, and shorter, but it is still awkward.

The value of s is defined in one function and used in another.

The **local** special form we introduced provides a natural way to bring the definition and use together.

```
(define (t-area4 a b c)
  (local
    ((define s (/ (+ a b c) 2)))
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

Full Racket provides several related language constructs. Each of the constructs has simpler semantics, but none is as general as **local**.

Since **local** is another form (like **cond**) that results in double parentheses, we will use square brackets to improve readability. This is another *convention*.

```
(define (t-area4 a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
```


Reusing names

Local definitions permit reuse of names.

This is not new to us:

```
(define n 10)  
(define (myfn n) (+ 2 n))  
(myfn 6)
```

gives the answer 8, not 12.

The substitution specified in the semantics of function application ensures that the correct value is used while evaluating the last line.

The name of a formal parameter to a function may reuse (within the body of that function) a name which is bound to a value through **define**.

Similarly, a **define** within a **local** expression may rebind a name which has already been bound to another value or expression.

The substitution rules we define for **local** as part of the semantic model must handle this.

Semantics of **local**

The substitution rule for **local** is the most complicated one we will see in this course.

It works by creating equivalent definitions that can be “promoted” to the top level.

An evaluation of **local** creates a **fresh** (new, unique) name for every name used in a local definition, binds the new name to the value, and substitutes the new name everywhere the old name is used in the expression.

Because the fresh names can't by definition appear anywhere outside the **local** expression, we can move the local definitions to the top level, evaluate them, and continue.

Before discussing the general case, we will demonstrate what happens in an application of our function **t-area4** which uses **local**.

In the example on the following slide, the local definition of **s** is rewritten using the fresh identifier **s_47**, which we just made up.

The Stepper does something similar in rewriting local identifiers, appending numbers to make them unique.

Evaluating t-area4

(t-area4 3 4 5) \Rightarrow

(local [(define s (/ (+ 3 4 5) 2))]

(sqrt (* s (− s 3) (− s 4) (− s 5)))) \Rightarrow

(define s_47 (/ (+ 3 4 5) 2))

(sqrt (* s_47 (− s_47 3) (− s_47 4) (− s_47 5))) \Rightarrow

(define s_47 (/ 12 2))

(sqrt (* s_47 (− s_47 3) (− s_47 4) (− s_47 5))) \Rightarrow

(define s_47 6)

(sqrt (* s_47 (− s_47 3) (− s_47 4) (− s_47 5))) \Rightarrow ... 6

In general, 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_new`) everywhere in the `local` expression.

The same thing is done with `x2` through `xn`.

The definitions `(define x1_new exp1) ... (define xn_new expn)` are then lifted out (all at once) to the top level of the program, preserving their ordering.

When all the rewritten definitions have been lifted out, what remains looks like (**local** [] **bodyexp**'), where **bodyexp**' is the rewritten version of **bodyexp**.

This is just replaced with **bodyexp**'. All of this (the renaming, the lifting, and removing the **local** with an empty definitions list) is a **single step**.

This is covered in Intermezzo 3 (Section 18), which you should read carefully. Make sure you understand the examples given there.

Naming 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 lst)**) occurs twice in the code.


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

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

In the function `eat-apples`, the subexpression

`(eat-apples (rest lst))`

appears in two different answers of the same `cond` expression, so only one of them will ever be evaluated.

Recall that in lecture module 07, we saw that a structurally-recursive version of `max-list` used the same recursive application twice.

We can use `local` to avoid this.

Old version of **max-list**

:: (max-list lon) produces the maximum element of lon

:: max-list: (listof Num) \rightarrow Num

:: requires: lon is nonempty

```
(define (max-list lon)
  (cond [(empty? (rest lon)) (first lon)]
        [(> (first lon) (max-list (rest lon))) (first lon)]
        [else (max-list (rest lon))]))
```

:: max-list2: (listof Num) \rightarrow Num

:: requires: lon is nonempty

(define (max-list2 lon) ; 2nd version

(cond [(empty? (rest lon)) (first lon)]

[else

(local [(define max-rest (max-list2 (rest lon)))]

(cond [(> (first lon) max-rest) (first lon)]

[else max-rest]))]))

This was also the reason that we used a helper function in the computation of a list of ancestors in an evolution tree.

Old version of **ancestors-ancient**

```
:: ancestors-ancient: Ancient Modern → (anyof (listof Str) false)
(define (ancestors-ancient from to)
  (cond
    [(cons? (ancestors (ancient-left from) to))
     (cons (ancient-name from) (ancestors (ancient-left from) to))]
    [(cons? (ancestors (ancient-right from) to))
     (cons (ancient-name from) (ancestors (ancient-right from) to))]
    [else false]))
```

We can now use **local** to similar effect.

```
(define (ancestors-ancient from to)
  (local [(define from-l (ancestors (ancient-left from) to))
          (define from-r (ancestors (ancient-right from) to))]
    (cond [(cons? from-l) (cons (ancient-name from) from-l)]
          [(cons? from-r) (cons (ancient-name from) from-r)]
          [else false])))
```


This new version of `ancestors-ancient` avoids making the same recursive function application twice, and does not require a helper function.

But it still suffers from an inefficiency: we always explore the entire evolution tree, even if the correct solution is found immediately in the left subtree.

We can avoid the extra search of the right subtree using nested `locals`.

```
(define (ancestors-ancient from to)
  (local [(define from-l (ancestors (ancient-left from) to))]
    (cond
      [(cons? from-l) (cons (ancient-name from) from-l)]
      [else
       (local [(define from-r (ancestors (ancient-right from) to))]
         (cond
           [(cons? from-r) (cons (ancient-name from) from-r)]
           [else false]))]))))
```

Using local for clarity

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.

Recall our function to compute the distance between two points.

```
(define (distance posn1 posn2)
  (sqrt (+ (sqr (— (posn-x posn1) (posn-x posn2)))
           (sqr (— (posn-y posn1) (posn-y posn2))))))
```

```
(define (distance posn1 posn2)
  (local [(define delta-x (— (posn-x posn1) (posn-x posn2)))
          (define delta-y (— (posn-y posn1) (posn-y posn2)))]
    (sqrt (+ (sqr delta-x) (sqr delta-y)))))
```

Encapsulation

Encapsulation is the process of grouping things together in a “capsule”.

We have already seen data encapsulation in the use of structures.

There is also an aspect of hiding information to encapsulation which we did not see with structures.

The local bindings are not visible (have no effect) outside the local expression.

In CS 136 we will see how objects combine data encapsulation with another type of encapsulation we now discuss.

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.

It keeps the “namespace” at the top level less cluttered, and makes the organization of the program more obvious.

This is particularly useful when using accumulators.

```
(define (sum-list lon)
  (local [(define (sum-list/acc lst sofar)
            (cond [(empty? lst) sofar]
                  [else (sum-list/acc (rest lst)
                                       (+ (first lst) sofar))])])
    (sum-list/acc lon 0)))
```

Making the accumulatively-recursive helper function local facilitates reasoning about the program.

HtDP (section VI) discusses justifying such code using an **invariant** which expresses the relationship between the arguments provided to the main function and the arguments to the helper function each time it is applied.

For summing a list, the invariant is that the sum of `lon` equals `sofar` plus the sum of `lst`.

This idea will be discussed further in CS 245. It is important in CS 240 and CS 341.

```
(define (isort lon)
  (local [(define (insert n slon)
            (cond [(empty? slon) (cons n empty)]
                  [(<= n (first slon)) (cons n slon)]
                  [else (cons (first slon) (insert n (rest slon)))]))]
    (cond [(empty? lon) empty]
          [else (insert (first lon) (isort (rest lon)))])))
```

Encapsulation and the design recipe

A function can enclose the cooperating helper functions that it uses inside a **local**, as long as these are not also needed by other functions. When this happens, the enclosing function and all the helpers act as a cohesive unit.

Here, the local helper functions require contracts and purposes, but not examples or tests. The helper functions can be tested by writing suitable tests for the enclosing function.

Make sure the local helper functions are still tested completely!

:: Full Design Recipe for isort ...

(define (isort lon)

(local [;; (insert n slon) inserts n into slon, preserving the order

;; insert: Num (listof Num) \rightarrow (listof Num)

;; requires: slon is sorted in nondecreasing order

(define (insert n slon)

(cond [(empty? slon) (cons n empty)]

[(\leq n (first slon)) (cons n slon)]

[else (cons (first slon) (insert n (rest slon)))]))])

(cond [(empty? lon) empty]

[else (insert (first lon) (isort (rest lon)))]))])

Terminology associated with local

The *binding occurrence* of a name is its use in a definition, or formal parameter to a function.

The associated *bound occurrences* are the uses of that name that correspond to that binding.

The *lexical scope* of a binding occurrence is all places where that binding has effect, taking note of holes caused by reuse of names.

Global scope is the scope of top-level definitions.

Making helper functions local can reduce the need to have parameters “go along for the ride”.

```
(define (countup-to n)  
  (countup-to-from n 0))
```

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

```
(define (countup2-to n)
  (local
    [(define (countup-from m)
      (cond [(> m n) empty]
            [else (cons m (countup-from (add1 m)))]))]
    (countup-from 0)))
```

Note that `n` no longer needs to be a parameter to `countup-from`, because it is in scope.

If we evaluate `(countup2-to 10)` using our substitution model, a renamed version of `countup-from` with `n` replaced by 10 is lifted to the top level.

Then, if we evaluate `(countup2-to 20)`, another renamed version of `countup-from` is lifted to the top level.

We can use the same idea to localize the helper functions for `mult-table` from lecture module 06.

Recall that

```
(mult-table 3 4) ⇒  
(list (list 0 0 0 0)  
      (list 0 1 2 3)  
      (list 0 2 4 6))
```

The c^{th} entry of the r^{th} row (numbering from 0) is $r \times c$.

:: code from lecture module 06

:: (mult-table nr nc) produces multiplication table

:: with nr rows and nc columns

:: mult-table: Nat Nat \rightarrow (listof (listof Nat))

```
(define (mult-table nr nc)
  (rows-from 0 nr nc))
```

:: (rows-from r nr nc) produces mult. table, rows r...(nr-1)

:: rows-from: Nat Nat Nat \rightarrow (listof (listof Nat))

```
(define (rows-from r nr nc)
  (cond [(>= r nr) empty]
        [else (cons (row r nc) (rows-from (add1 r) nr nc))]))
```

:: (row r nc) produces rth row of mult. table of length nc

:: row: Nat Nat \rightarrow (listof Nat)

```
(define (row r nc)  
  (cols-from 0 r nc))
```

:: (cols-from c r nc) produces entries c...(nc-1) of rth row of mult. table

:: cols-from: Nat Nat Nat \rightarrow (listof Nat)

```
(define (cols-from c r nc)  
  (cond [( $\geq$  c nc) empty]  
        [else (cons (* r c) (cols-from (add1 c) r nc))]))
```

```
(define (mult-table2 nr nc)
  (local
    [(define (row r)
      (local [(define (cols-from c)
                  (cond [(>= c nc) empty]
                        [else (cons (* r c) (cols-from (add1 c)))]))]
        (cols-from 0)))
      (define (rows-from r)
        (cond [(>= r nr) empty]
              [else (cons (row r) (rows-from (add1 r)))]))]
      (rows-from 0)))
```

If we evaluate `(mult-table2 3 4)` using the substitution model, the outermost `local` is evaluated once.

But `(row r)` is evaluated four times, for $r = 0, 1, 2, 3$.

This means that the innermost `local` is evaluated four times, and four renamed versions of `cols-from` are lifted to the top level, each with a different value of `r` substituted.

We will further simplify this code in lecture module 10.

The use of **local** has permitted only modest gains in expressivity and readability in our examples.

The language features discussed in the next module expand this power considerably.

Some other languages (C, C++, Java) either disallow nested function definitions or allow them only in very restricted circumstances.

Local variable and constant definitions are more common.

Goals of this module

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 understand the idea of encapsulation of local helper functions.

You should be able to match the use of any constant or function name in a program to the binding to which it refers.

Functional abstraction

Readings: HtDP, sections 19-24.

Language level: Intermediate Student With Lambda

- different order used in lecture
- section 24 material introduced much earlier
- sections 22, 23 not covered in lecture

What is abstraction?

Abstraction is the process of finding similarities or common aspects, and forgetting unimportant differences.

Example: writing a function.

The differences in parameter values are forgotten, and the similarity is captured in the function body.

We have seen many similarities between functions, and captured them in design recipes.

But some similarities still elude us.

Eating apples

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

Keeping odd numbers

```
(define (keep-odds lst)
  (cond [(empty? lst) empty]
        [(odd? (first lst))
         (cons (first lst) (keep-odds (rest lst)))]
        [else (keep-odds (rest lst))]))
```

Abstracting from these examples

What these two functions have in common is their general structure.

Where they differ is in the specific predicate used to decide whether an item is removed from the answer or not.

We could write one function to do both these tasks if we could supply, as an argument to that function, the predicate to be used.

The Intermediate language permits this.

In the Intermediate language, functions are values. In fact, they are *first-class* values.

Functions have the same status as the other values we've seen.

They can be:

1. consumed as function arguments
2. produced as function results
3. bound to identifiers
4. put in structures and lists

This is a feature that many mainstream programming languages lack but is central to functional programming languages.

More rapidly-evolving or recently-defined languages that are not primarily functional (e.g. Python, Ruby, Perl 6, C#) do implement this feature to some extent.

Java and C++ provide other abstraction mechanisms that provide some of the benefits.

Functions-as-values provides a clean way to think about the concepts and issues involved in abstraction.

You can then worry about how to implement a high-level design in a given programming language.

Consuming functions

```
(define (foo f x y) (f x y))
```

```
(foo + 2 3)  $\Rightarrow$  5
```

```
(foo * 2 3)  $\Rightarrow$  6
```

my-filter

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


Tracing my-filter

`(my-filter odd? (list 5 6 7))`

\Rightarrow `(cons 5 (my-filter odd? (list 6 7)))`

\Rightarrow `(cons 5 (my-filter odd? (list 7)))`

\Rightarrow `(cons 5 (cons 7 (my-filter odd? empty)))`

\Rightarrow `(cons 5 (cons 7 empty))`

`my-filter` is an **abstract list function** which handles the general operation of removing items from lists.

Using my-filter

```
(define (keep-odds lst) (my-filter odd? lst))
```

```
(define (not-symbol-apple? item) (not (symbol=? item 'apple)))
```

```
(define (eat-apples lst) (my-filter not-symbol-apple? lst))
```

The function `filter`, which behaves identically to our `my-filter`, is built into Intermediate Student and full Racket.

`filter` and other abstract list functions provided in Racket are used to apply common patterns of structural recursion.

We'll discuss how to write contracts for them shortly.

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.

Producing Functions

We saw in lecture module 09 how **local** could be used to create functions during a computation, to be used in evaluating the body of the **local**.

But now, because functions are values, the body of the **local** can produce such a function as a value.

Though it is not apparent at first, this is enormously useful.

We illustrate with a very small example.

```
(define (make-adder n)
  (local
    [(define (f m) (+ n m))]
    f))
```

What is `(make-adder 3)`?

We can answer this question with a trace.

`(make-adder 3) ⇒`

`(local [(define (f m) (+ 3 m))] f) ⇒`

`(define (f_42 m) (+ 3 m)) f_42`

`(make-adder 3)` is the renamed function `f_42`, which is a function that adds 3 to its argument.

We can apply this function immediately, or we can use it in another expression, or we can put it in a data structure.

Here's what happens if we apply it immediately.

`((make-adder 3) 4) \Rightarrow`

`((local [(define (f m) (+ 3 m))] f) 4) \Rightarrow`

`(define (f_42 m) (+ 3 m)) (f_42 4) \Rightarrow`

`(+ 3 4) \Rightarrow 7`

Binding Functions to Identifiers

The result of `make-adder` can be bound to an identifier and then used repeatedly.

```
(define my-add1 (make-adder 1)) ; add1 is a built-in function
```

```
(define add3 (make-adder 3))
```

```
(my-add1 3)  $\Rightarrow$  4
```

```
(my-add1 10)  $\Rightarrow$  11
```

```
(add3 13)  $\Rightarrow$  16
```


How does this work?

`(define my-add1 (make-adder 1))` \Rightarrow

`(define my-add1 (local [(define (f m) (+ 1 m))] f))` \Rightarrow

`(define (f_43 m) (+ 1 m))` ; rename and lift out f

`(define my-add1 f_43)`

`(my-add1 3)` \Rightarrow

`(f_43 3)` \Rightarrow

`(+ 1 3)` \Rightarrow

4

Putting functions in lists

Recall our code in lecture module 08 for evaluating alternate arithmetic expressions such as `'(+ (* 3 4) 2)`.

`:: eval: AltAExp → Num`

```
(define (eval aax)  
  (cond [(number? aax) aax]  
        [else (my-apply (first aax) (rest aax))]))
```

:: my-apply: Sym AltAExpList \rightarrow Num

```
(define (my-apply f aaxl)
  (cond [(and (empty? aaxl) (symbol=? f '*)) 1]
        [(and (empty? aaxl) (symbol=? f '+)) 0]
        [(symbol=? f '*)
         (* (eval (first aaxl)) (my-apply f (rest aaxl)))]
        [(symbol=? f '+)
         (+ (eval (first aaxl)) (my-apply f (rest aaxl)))]))
```

Note the similar-looking code.

Much of the code is concerned with translating the symbol `'+` into the function `+`, and the same for `'*` and `*`.

If we want to add more functions to the evaluator, we have to write more code which is very similar to what we've already written.

We can use an association list to store the above correspondence, and use the function `lookup-al` we saw in lecture module 08 to look up symbols.

```
(define trans-table (list (list '+ +)  
                           (list '* *)))
```

Now (lookup-al '+ trans-table) produces the function +.

((lookup-al '+ trans-table) 3 4 5) \Rightarrow 12

$:: \text{newapply}: \text{Sym AltAExpList} \rightarrow \text{Num}$

```
(define (newapply f aaxl)
  (cond [(and (empty? aaxl) (symbol=? f '*)) 1]
        [(and (empty? aaxl) (symbol=? f '+)) 0]
        [else ((lookup-al f trans-table)
                 (eval (first aaxl)) (newapply f (rest aaxl))))])
```

We can simplify this even further, because in Intermediate Student, $+$ and $*$ allow zero arguments.

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

:: newapply: Sym AltAExpList \rightarrow Num

```
(define (newapply f axl)
  (local [(define op (lookup-al f trans-table))]
    (cond [(empty? axl) (op)]
          [else (op (eval (first axl))
                          (newapply f (rest axl)))])))
```

Now, to add a new binary function (that is also defined for 0 arguments), we need only add one line to [trans-table](#).

Contracts and types

Our contracts describe the type of data consumed by and produced by a function.

Until now, the type of data was either a basic (built-in) type, a defined (struct) type, an `anyof` type, or a list type, such as List-of-Symbols, which we then called (listof Sym).

Now we need to talk about the type of a function consumed or produced by a function.

We can use the contract for a function as its type.

For example, the type of `>` is $(\text{Num Num} \rightarrow \text{Bool})$, because that's the contract of that function.

We can then use type descriptions of this sort in contracts for functions which consume or produce other functions.

Simulating Structures

We can use the ideas of producing and binding functions to simulate structures.

```
(define (my-make-posn x y)
  (local
    [(define (symbol-to-value s)
      (cond [(symbol=? s 'x) x]
            [(symbol=? s 'y) y]))]
    symbol-to-value))
```

A trace demonstrates how this function works.

```
(define p1 (my-make-posn 3 4)) ⇒  
(define p1 (local  
  [(define (symbol-to-value s)  
    (cond [(symbol=? s 'x) 3]  
          [(symbol=? s 'y) 4]))]  
  symbol-to-value))
```

Notice how the parameters have been substituted into the **local** definition.

We now rename **symbol-to-value** and lift it out.

This yields:

```
(define (symbol-to-value_38 s)
  (cond [(symbol=? s 'x) 3]
        [(symbol=? s 'y) 4]))
(define p1 symbol-to-value_38)
```

`p1` is now a function with the `x` and `y` values we supplied to `my-make-posn` coded in.

To get out the `x` value, we can use `(p1 'x)`:

$$(p1 \text{ 'x}) \Rightarrow 3$$

We can define a few convenience functions to simulate `posn-x` and `posn-y`:

```
(define (my-posn-x p) (p 'x))
```

```
(define (my-posn-y p) (p 'y))
```

If we apply `my-make-posn` again with different values, it will produce a different rewritten and lifted version of `symbol-to-value`, say `symbol-to-value_39`.

We have just seen how to implement structures without using lists.

Scope revisited

```
(define (my-make-posn x y)
  (local
    [(define (symbol-to-value s)
      (cond [(symbol=? s 'x) x]
            [(symbol=? s 'y) y]))]
    symbol-to-value))
```

Consider the use of `x` inside `symbol-to-value`.

Its binding occurrence is outside the definition of `symbol-to-value`.

Our trace made it clear that the result of a particular application, say `(my-make-posn 3 4)`, is a “copy” of `symbol-to-value` with 3 and 4 substituted for `x` and `y`, respectively.

That “copy” can be used much later, to retrieve the value of `x` or `y` that was supplied to `my-make-posn`.

This is possible because the “copy” of `symbol-to-value`, even though it was defined in a `local` definition, survives after the evaluation of the `local` is finished.

Anonymous functions

```
(local  
  [(define (symbol-to-value s)  
    (cond [(symbol=? s 'x) x]  
          [(symbol=? s 'y) y]))]  
  symbol-to-value)
```

The result of evaluating this expression is a function.

What is its name? It is **anonymous** (has no name).

This is sufficiently valuable that there is a special mechanism for it.

Producing anonymous functions

```
(define (not-symbol-apple? item) (not (symbol=? item 'apple)))  
(define (eat-apples lst) (filter not-symbol-apple? lst))
```

This is a little unsatisfying, because `not-symbol-apple?` is such a small and relatively useless function.

It is unlikely to be needed elsewhere.

We can avoid cluttering the top level with such definitions by putting them in `local` expressions.

```
(define (eat-apples lst)
  (local [(define (not-symbol-apple? item)
              (not (symbol=? item 'apple)))]
    (filter not-symbol-apple? lst)))
```

This is as far as we would go based on our experience with **local**.

But now that we can use functions as values, the value produced by the local expression can be the function **not-symbol-apple?**.

We can then take that value and deliver it as an argument to **filter**.

```
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                     (not (symbol=? item 'apple)))]
            not-symbol-apple?)
    lst))
```

But this is still unsatisfying. Why should we have to name `not-symbol-apple?` at all? In the expression `(* (+ 2 3) 4)`, we didn't have to name the intermediate value 5.

Racket provides a mechanism for constructing a nameless function which can then be used as an argument.

Introducing lambda

```
(local [(define (name-used-once x1 ... xn) exp)]  
  name-used-once)
```

can also be written

```
(lambda (x1 ... xn) exp)
```

lambda can be thought of as “make-function”.

It can be used to create a function which we can then use as a value
– for example, as the value of the first argument of **filter**.

We can then replace

```
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                      (not (symbol=? item 'apple)))]
            not-symbol-apple?)
    lst))
```

with the following:

```
(define (eat-apples lst)
  (filter (lambda (item) (not (symbol=? item 'apple))) lst))
```

lambda is available in Intermediate Student with Lambda, and discussed in section 24 of the textbook.

We're jumping ahead to it because of its central importance in Racket, Lisp, and the history of computation in general.

The designers of the teaching languages could have renamed it as they did with other constructs, but chose not to out of respect.

The word **lambda** comes from the Greek letter, used as notation in the first formal model of computation.

We can use **lambda** to simplify our implementation of **posn** structures.

```
(define (my-make-posn x y)
  (lambda (s)
    (cond [(symbol=? s 'x) x]
          [(symbol=? s 'y) y])))
```

lambda also underlies the definition of functions.

Until now, we have had two different types of definitions.

;; a definition of a numerical constant

```
(define interest-rate 3/100)
```

;; a definition of a function to compute interest

```
(define (interest-earned amount)  
  (* interest-rate amount))
```

There is really only one kind of **define**, which binds a name to a value.

Internally,

```
(define (interest-earned amount)  
  (* interest-rate amount))
```

is translated to

```
(define interest-earned  
  (lambda (amount) (* interest-rate amount)))
```

which binds the name `interest-earned` to the value

```
(lambda (amount) (* interest-rate amount)).
```

We should change our semantics for function definition to represent this rewriting.

But doing so would make traces much harder to understand.

As long as the value of defined constants (now including functions) cannot be changed, we can leave their names unsubstituted in our traces for clarity.

But in CS 136, when we introduce mutation, we will have to make this change.

Syntax and semantics of Intermediate Student with Lambda

We don't have to make many changes to our earlier syntax and semantics.

The first position in an application can now be an expression (computing the function to be applied).

If this is the case, it must be evaluated along with the other arguments.

Before, the next thing after the open parenthesis in a function application had to be a defined or primitive function name.

Now a function application can have two or more open parentheses in a row, as in `((make-adder 3) 4)`.

We need a rule for evaluating applications where the function being applied is anonymous (a **lambda** expression.)

$$((\text{lambda } (x_1 \dots x_n) \text{ exp}) v_1 \dots v_n) \Rightarrow \text{exp}'$$

where exp' is exp with all occurrences of x_1 replaced by v_1 , all occurrences of x_2 replaced by v_2 , and so on.

As an example:

$$((\text{lambda } (x \ y) (* (+ y 4) x)) 5 \ 6) \Rightarrow (* (+ 6 4) 5)$$

Here's `make-adder` rewritten using `lambda`.

```
(define make-adder  
  (lambda (x)  
    (lambda (y)  
      (+ x y))))
```

We see the same scope phenomenon we discussed earlier: the binding occurrence of `x` lies outside `(lambda (y) (+ x y))`.

What is `(make-adder 3)`?

`(make-adder 3) ⇒`

`((lambda (x) (lambda (y) (+ x y))) 3) ⇒`

`(lambda (y) (+ 3 y))`

This shorthand is very useful, which is why `lambda` is being added to mainstream programming languages.

Suppose during a computation, we want to specify some action to be performed one or more times in the future.

Before knowing about **lambda**, we might build a data structure to hold a description of that action, and a helper function to consume that data structure and perform the action.

Now, we can just describe the computation clearly using **lambda**.

Example: character translation in strings

Recall that Racket provides the function `string→list` to convert a string to a list of characters.

This is the most effective way to work with strings, though typically structural recursion on these lists is not natural, and generative recursion (as discussed in module 11) is used.

In the example we are about to discuss, structural recursion works.

We'll develop a general method of performing character translations on strings.

For example, we might want to convert every 'a' in a string to a 'b'.

The string "abracadabra" becomes "bbrbcdbbbrb".

This doesn't require functional abstraction. You could have written a function that does this in lecture module 05.

:: $a \rightarrow b$: $\text{Str} \rightarrow \text{Str}$

(define ($a \rightarrow b$ str)

(list \rightarrow string ($a \rightarrow b$ /loc (string \rightarrow list str))))

:: $a \rightarrow b$ /loc: (listof Char) \rightarrow (listof Char)

(define ($a \rightarrow b$ /loc loc)

(cond [(empty? loc) empty]

[(char=? (first loc) #\a) (cons #\b ($a \rightarrow b$ /loc (rest loc)))]

[else (cons (first loc) ($a \rightarrow b$ /loc (rest loc)))]))

Functional abstraction is useful in generalizing this example.

Here, we went through a string applying a predicate (“equals a?”) to each character, and applied an action (“make it b”) to characters that satisfied the predicate.

A translation is a pair (a list of length two) consisting of a predicate and an action.

We might want to apply several translations to a string.

We can describe the translation in our example like this:

```
(list (lambda (c) (char=? #\a c))  
      (lambda (c) #\b))
```

Since these are likely to be common sorts of functions, we can write helper functions to create them.

```
(define (is-char? c1) (lambda (c2) (char=? c1 c2)))  
(define (always c1) (lambda (c2) c1))
```

```
(list (is-char? #\a)  
      (always #\b))
```

Our `translate` function will consume a list of translations and a string to be translated.

For each character `c` in the string, it will create a result character by applying the action of the first translation on the list whose predicate is satisfied by `c`.

If no predicate is satisfied by `c`, the result character is `c`.

An example of its use: suppose we have a string `s`, and we want a version of it where all letters are capitalized, and all numbers are “redacted” by replacing them with asterisks.

```
(define s "Testing 1-2-3.")  
(translate (list (list char-alphabetic? char-upcase)  
                (list char-numeric? (always #\*)))  
            s)
```

⇒ "TESTING *-*-*."

`char-alphabetic?`, `char-upcase`, and `char-numeric?` are primitive functions in the teaching languages.

```
(define (translate lot str) (list→string (trans-loc lot (string→list str))))
```

```
(define (trans-loc lot loc)  
  (cond [(empty? loc) empty]  
        [else (cons (trans-char lot (first loc))  
                     (trans-loc lot (rest loc)))]))
```

```
(define (trans-char lot c)  
  (cond [(empty? lot) c]  
        [((first (first lot)) c) ((second (first lot)) c)]  
        [else (trans-char (rest lot) c)]))
```

What is the contract for `trans-loc`?

Contracts for abstract list functions

`filter` consumes a function and a list, and produces a list.

We might be tempted to conclude that its contract is

$(\text{Any} \rightarrow \text{Bool}) (\text{listof Any}) \rightarrow (\text{listof Any})$.

But this is not specific enough.

The application `(filter odd? (list 'a 'b 'c))` does not violate the above contract, but will clearly cause an error.

There is a relationship among the two arguments to `filter` and the result of `filter` that we need to capture in the contract.

Parametric types

An application `(filter pred? lst)`, can work on any type of list, but the predicate provided should consume elements of that type of list.

In other words, we have a dependency between the type of the predicate (which is the contract of the predicate) and the type of list.

To express this, we use a type variable, such as X , and use it in different places to indicate where the same type is needed.

The contract for filter

`filter` consumes a predicate with contract $X \rightarrow \text{Bool}$, where X is the base type of the list that it also consumes.

It produces a list of the same type it consumes.

The contract for `filter` is thus:

$:: \text{filter}: (X \rightarrow \text{Bool}) (\text{listof } X) \rightarrow (\text{listof } X)$

Here X stands for the unknown data type of the list.

We say `filter` is *polymorphic* or *generic*; it works on many different types of data.

The contract for **filter** has three occurrences of a type variable X .

Since a type variable is used to indicate a relationship, it needs to be used at least twice in any given contract.

A type variable used only once can probably be replaced with **Any**.

We will soon see examples where more than one type variable is needed in a contract.

Understanding contracts

Many of the difficulties one encounters in using abstract list functions can be overcome by careful attention to contracts.

For example, the contract for the function provided as an argument to `filter` says that it consumes one argument and produces a Boolean value.

This means we must take care to never use `filter` with an argument that is a function that consumes two variables, or that produces a number.

Abstracting from examples

Here are two early list functions we wrote.

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

```
(define (compute-taxes lst)
  (cond [(empty? lst) empty]
        [else (cons (sr→tr (first lst))
                     (compute-taxes (rest lst)))]))
```

We look for a difference that can't be explained by renaming (it being what is applied to the first item of a list) and make that a parameter.

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

Tracing **my-map**

`(my-map sqr (list 3 6 5))`

\Rightarrow `(cons 9 (my-map sqr (list 6 5)))`

\Rightarrow `(cons 9 (cons 36 (my-map sqr (list 5))))`

\Rightarrow `(cons 9 (cons 36 (cons 25 (my-map sqr empty))))`

\Rightarrow `(cons 9 (cons 36 (cons 25 empty)))`

my-map performs the general operation of transforming a list element-by-element into another list of the same length.

The application

`(my-map f (list x1 x2 ... xn))` has the same effect as evaluating
`(list (f x1) (f x2) ... (f xn))`.

We can use `my-map` to give short definitions of a number of functions we have written to consume lists:

`(define (negate-list lst) (my-map — lst))`

`(define (compute-taxes lst) (my-map sr→tr lst))`

How can we use `my-map` to rewrite `trans-loc`?

The contract for my-map

`my-map` consumes a function and a list, and produces a list.

How can we be more precise about its contract, using parametric type variables?

Built-in abstract list functions

Intermediate Student also provides `map` as a built-in function, as well as many other abstract list functions.

There is a useful table on page 313 of the text (Figure 57 in section 21.2).

The abstract list functions `map` and `filter` allow us to quickly describe functions to do something to all elements of a list, and to pick out selected elements of a list, respectively.

Building an abstract list function

The functions we have worked with so far consume and produce lists.

What about abstracting from functions such as `count-symbols` and `sum-of-numbers`, which consume lists and produce values?

Let's look at these, find common aspects, and then try to generalize from the template.

```
(define (sum-of-numbers lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum-of-numbers (rest lst)))]))
```

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

```
(define (count-symbols lst)
  (cond [(empty? lst) 0]
        [else (+ 1 (count-symbols (rest lst)))]))
```

Note that each of these examples has a base case which is a value to be returned when the argument list is **empty**.

Each example is applying some function to combine **(first lst)** and the result of a recursive function application with argument **(rest lst)** .

This continues to be true when we look at the list template and generalize from that.

```
(define (my-list-fn lst)
  (cond [(empty? lst) ...]
        [else (... (first lst) ...
                     (my-list-fn (rest lst)) ... )]))
```

We replace the first ellipsis by a base value.

We replace the rest of the ellipses by some function which combines `(first lst)` and the result of a recursive function application on `(rest lst)`.

This suggests passing the base value and the combining function as parameters to an abstract list function.

The abstract list function **foldr**

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

foldr is also a built-in function in Intermediate Student With Lambda.

Tracing my-foldr

$(\text{my-foldr } f \ 0 \ (\text{list } 3 \ 6 \ 5)) \Rightarrow$
 $(f \ 3 \ (\text{my-foldr } f \ 0 \ (\text{list } 6 \ 5))) \Rightarrow$
 $(f \ 3 \ (f \ 6 \ (\text{my-foldr } f \ 0 \ (\text{list } 5)))) \Rightarrow$
 $(f \ 3 \ (f \ 6 \ (f \ 5 \ (\text{my-foldr } f \ 0 \ \text{empty})))) \Rightarrow$
 $(f \ 3 \ (f \ 6 \ (f \ 5 \ 0))) \Rightarrow \dots$

Intuitively, the effect of the application

$(\text{foldr } f \ b \ (\text{list } x1 \ x2 \ \dots \ xn))$ is to compute the value of the expression
 $(f \ x1 \ (f \ x2 \ (\dots (f \ xn \ b) \ \dots)))$.

`foldr` is short for “fold right”.

The reason for the name is that it can be viewed as “folding” a list using the provided `combine` function, starting from the right-hand end of the list.

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

The contract for foldr

`foldr` consumes three arguments:

- a function which combines the first list item with the result of reducing the rest of the list;
- a base value;
- a list on which to operate.

What is the contract for `foldr`?

Using foldr

```
(define (sum-of-numbers lst) (foldr + 0 lst))
```

If `lst` is `(list x1 x2 ... xn)`, then by our intuitive explanation of `foldr`, the expression `(foldr + 0 lst)` reduces to

```
(+ x1 (+ x2 (+ ... (+ xn 0) ...)))
```

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

The function provided to `foldr` consumes two parameters: one is an element on the list which is an argument to `foldr`, and one is the result of reducing the rest of the list.

Sometimes one of those arguments should be ignored, as in the case of using `foldr` to compute `count-symbols`.

The important thing about the first argument to the function provided to `foldr` is that it 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 reduction of the rest of the list.

```
(define (count-symbols lst) (foldr (lambda (x y) (add1 y)) 0 lst))
```

The function provided to `foldr`, namely

`(lambda (x y) (add1 y))`, ignores its first argument.

Its second argument represents the reduction of the rest of the list (in this case the length of the rest of the list, to which 1 must be added).

Using foldr to produce lists

So far, the functions we have been providing to `foldr` have produced numerical results, but they can also produce `cons` expressions.

`foldr` is an abstraction of structural recursion on lists, so we should be able to use it to implement `negate-list` from module 05.

We need to define a function `(lambda (x y) ...)` where `x` is the first element of the list and `y` is the result of the recursive function application.

`negate-list` takes this element, negates it, and `conses` it onto the result of the recursive function application.

The function we need is

```
(lambda (x y) (cons (— x) y))
```

Thus we can give a nonrecursive version of `negate-list` (that is, `foldr` does all the recursion).

```
(define (negate-list lst)
  (foldr (lambda (x y) (cons (— x) y)) empty lst))
```

Because we generalized `negate-list` to `map`, we should be able to use `foldr` to define `map`.

Let's look at the code for `my-map`.

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

Clearly `empty` is the base value, and the function provided to `foldr` is something involving `cons` and `f`.

In particular, the function provided to `foldr` must apply `f` to its first argument, then `cons` the result onto its second argument (the reduced rest of the list).

```
(define (my-map f lst)
  (foldr (lambda (x y) (cons (f x) y)) empty lst))
```

We can also implement `my-filter` using `foldr`.

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 (unless it ends the recursion early, like `insert`).

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.

Higher-order functions

Functions that consume or produce functions like `filter`, `map`, and `foldr` are sometimes called **higher-order functions**.

Another example is the built-in `build-list`. This consumes a natural number `n` and a function `f`, and produces the list

```
(list (f 0) (f 1) ... (f (sub1 n)))
```

```
(build-list 4 add1)  $\Rightarrow$  (list 1 2 3 4).
```

Clearly `build-list` abstracts the “count up” pattern, and it is easy to write our own version.

```
(define (my-build-list n f)
  (local
    [(define (list-from i)
      (cond [(>= i n) empty]
            [else (cons (f i) (list-from (add1 i)))]))]
    (list-from 0)))
```

We can now simplify `mult-table` even further.

```
(define (mult-table nr nc)
  (build-list nr
    (lambda (i)
      (build-list nc
        (lambda (j)
          (* i j)))))))
```


Goals of this module

You should understand the idea of functions as first-class values: how they can be supplied as arguments, produced as values using **lambda**, bound to identifiers, and placed in lists.

You should be familiar with the built-in list functions provided by Racket, understand how they abstract common recursive patterns, and be able to use them to write code.

You should be able to write your own abstract list functions that implement other recursive patterns.

You should understand how to do step-by-step evaluation of programs written in the Intermediate language that make use of functions as values.

Generative and accumulative recursion

Readings: Sections 25, 26, 27, 30, 31

- Some subsections not explicitly covered in lecture
- Section 27.2 technique applied to strings

What is generative recursion?

Structural recursion, which we have been using so far, is a way of deriving code whose form parallels a data definition.

Generative recursion is more general: the recursive cases are **generated** based on the problem to be solved.

The non-recursive cases also do not follow from a data definition.

It is much harder to come up with such solutions to problems.

It often requires deeper analysis and domain-specific knowledge.

Example revisited: GCD

:: (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm

:: euclid-gcd: Nat Nat \rightarrow Nat

```
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

Why does this work?

Correctness: Follows from Math 135 proof of the identity.

Termination: An application terminates if it can be reduced to a value in finite time.

All of our functions so far have terminated. But why?

For a non-recursive function, it is easy to argue that it terminates, assuming all applications inside it do.

It is not clear what to do for recursive functions.

Termination of recursive functions

Why did our structurally recursive functions terminate?

A structurally recursive function always makes recursive applications on smaller instances, whose size is bounded below by the base case (e.g. the empty list).

We can thus bound the **depth** of recursion (the number of applications of the function before arriving at a base case).

As a result, the evaluation cannot go on forever.

`(sum-list (list 3 6 5 4)) ⇒`
`(+ 3 (sum-list (list 6 5 4))) ⇒`
`(+ 3 (+ 6 (sum-list (list 5 4)))) ⇒ ...`

The depth of recursion of any application of `sum-list` is equal to the length of the list to which it is applied.

For generatively recursive functions, we need to make a similar argument.

In the case of `euclid-gcd`, our measure of progress is the size of the second argument.

If the first argument is smaller than the second argument, the first recursive application switches them.

After that, the second argument is always smaller than the first argument in any recursive application.

The second argument always gets smaller in the recursive application (since $m > n \bmod m$), but it is bounded below by 0.

Thus any application of `euclid-gcd` has a depth of recursion bounded by the second argument.

In fact, it is always much faster than this.

Termination is sometimes hard

```
:: collatz: Nat → Nat
```

```
:: requires: n ≥ 1
```

```
(define (collatz n)  
  (cond [(= n 1) 1]  
        [(even? n) (collatz (/ n 2))]  
        [else (collatz (+ 1 (* 3 n)))]))
```

It is a decades-old open research problem to discover whether or not `(collatz n)` terminates for all values of `n`.

We can see better what `collatz` is doing by producing a list.

`:: (collatz-list n)` produces the list of the intermediate

`::` results calculated by the `collatz` function.

`:: collatz-list: Nat \rightarrow (listof Nat)`

`:: requires: n \geq 1`

`(check-expect (collatz-list 1) '(1))`

`(check-expect (collatz-list 4) '(4 2 1))`

`(define (collatz-list n)`

`(cons n (cond [(= n 1) empty]`

`[(even? n) (collatz-list (/ n 2))]`

`[else (collatz-list (+ 1 (* 3 n)))])))`

Hoare's Quicksort

The Quicksort algorithm is an example of **divide and conquer**:

- divide a problem into smaller subproblems;
- recursively solve each one;
- combine the solutions to solve the original problem.

Quicksort sorts a list of numbers into non-decreasing order by first choosing a **pivot** element from the list.

The subproblems consist of the elements less than the pivot, and those greater than the pivot.

If the list is (list 9 4 15 2 12 20), and the pivot is 9, then the subproblems are (list 4 2) and (list 15 12 20).

Recursively sorting the two subproblem lists gives (list 2 4) and (list 12 15 20).

It is now simple to combine them with the pivot to give the answer.

(append (list 2 4) (list 9) (list 12 15 20))

The easiest pivot to select from a list `lon` is `(first lon)`.

A function which tests whether another item is less than the pivot is `(lambda (x) (< x (first lon)))`.

The first subproblem is then

`(filter (lambda (x) (< x (first lon))) lon)`.

A similar expression will find the second subproblem (items greater than the pivot).

:: (quick-sort lon) sorts lon in non-decreasing order

:: quick-sort: (listof Num) \rightarrow (listof Num)

```
(define (quick-sort lon)
  (cond [(empty? lon) empty]
        [else (local
                  [(define pivot (first lon))
                   (define less (filter (lambda (x) (< x pivot)) (rest lon)))
                   (define greater (filter (lambda (x) (>= x pivot)) (rest lon)))]
                (append (quick-sort less) (list pivot) (quick-sort greater))))])
```


Termination of quicksort follows from the fact that both subproblems have fewer elements than the original list (since neither contains the pivot).

Thus the depth of recursion of an application of `quick-sort` is bounded above by the number of elements in the argument list.

This would not have been true if we had mistakenly written

```
(filter (lambda (x) (>= x pivot)) lon)
```

The teaching languages function `quicksort` (note no hyphen) consumes two arguments, a list and a comparison function.

`(quicksort '(1 5 2 4 3) <)` \Rightarrow `'(1 2 3 4 5)`

`(quicksort '(1 5 2 4 3) >)` \Rightarrow `'(5 4 3 2 1)`

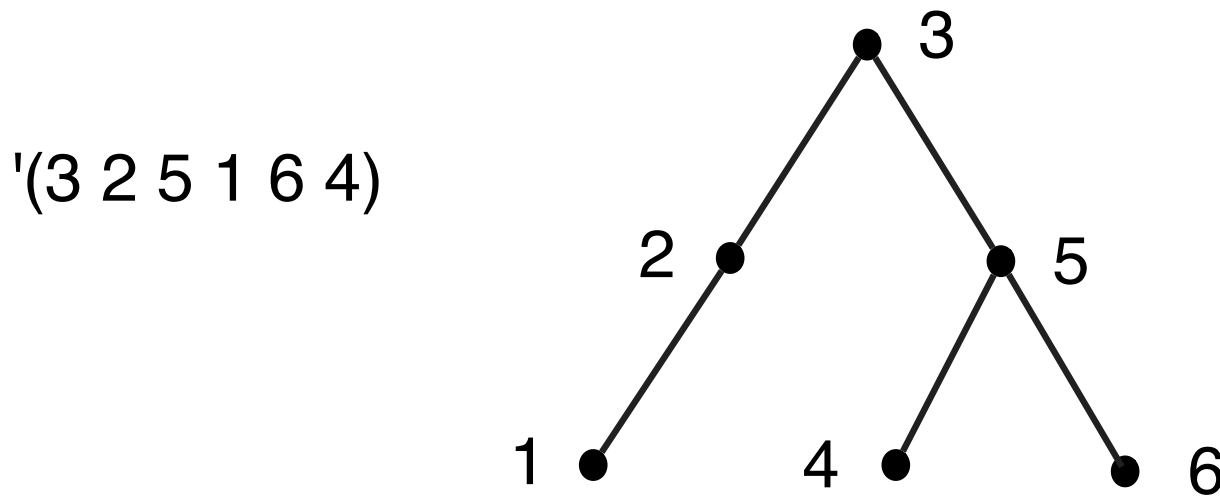
Intuitively, quicksort works best when the two recursive function applications are on arguments about the same size.

When one recursive function application is always on an empty list (as is the case when **quick-sort** is applied to an already-sorted list), the pattern of recursion is similar to the worst case of insertion sort, and the number of steps is roughly proportional to the square of the length of the list.

We will go into more detail on efficiency considerations in CS 136.

Here is another way of looking at quicksort.

Consider taking a list and inserting each element into an initially-empty binary search tree as a key with some dummy value, using an accumulator so that the first element of the list is the first inserted.



```

(define (list→bst lon)
  (local [(define (list→bst/acc lon bst-so-far)
            (cond [(empty? lon) bst-so-far]
                  [else (list→bst/acc
                           (rest lon)
                           (add-to-bst (first lon)
                                       "dummy"
                                       bst-so-far))]))])
    (list→bst/acc lon empty)))

```

Inserting the elements of the list in order uses structural recursion on the list, with an accumulator for the BST.

Each insertion uses structural recursion on the BST.

We can recover the original list, but in sorted order, by flattening the tree in such a manner that the key of the root node comes between the flattened result of the left tree and the flattened result of the right tree (instead of before both).

This is called an **inorder traversal**, and computing it also uses structural recursion on the BST.

:: (bst-inorder t) flattens a BST into a sorted list

:: with root between left and right

:: bst-inorder: BST \rightarrow (listof Num)

```
(define (bst-inorder t)
  (cond [(empty? t) empty]
        [else (append (bst-inorder (node-left t))
                        (list (node-key t))
                        (bst-inorder (node-right t))))])
```

```
(define (treesort lon)
  (bst-inorder (list→bst lon)))
```

Quicksort can be viewed as simply eliminating the trees from treesort.

The first key inserted into the BST, which ends up at the root, is the pivot.

(bst-inorder (node-left t)) is exactly the list of items of value less than the pivot.

Quicksort takes a number of structurally-recursive computations and makes them more efficient by using generative recursion.

Modifying the design recipe

The design recipe becomes much more vague when we move away from data-directed design.

The purpose statement should now specify **how** the function works, as well as what it does.

Examples need to illustrate the workings of the algorithm.

The template is less useful. Typically there are tests for the easy cases that don't require recursion, followed by the formulation and recursive solution of subproblems, and then combination of the solutions.

Example: breaking strings into lines

Traditionally, the character set used in computers has included not only alphanumeric characters and punctuation, but “control” characters as well.

An example in Racket is `#\newline`, which signals the start of a new line of text.

The characters ‘\’ and ‘n’ appearing consecutively in a string constant are interpreted as a single newline character.

For example, the string `"ab\n cd"` is a five-character string with a newline as the third character.

A line is a maximal sequence of non-newline characters.

Text files often contain embedded newlines to split the file into lines.

The same can be true of strings entered by a user or displayed.

We will develop a function to break a string into lines.

```
(check-expect (string→strlines "abc\ndef") (list "abc" "def"))
```

What should the following produce?

```
(string→strlines "abc\ndef\nhig")
```

```
(string→strlines "abc\n\nhig")
```

```
(string→strlines "\nabc")
```

```
(string→strlines " ")
```

```
(string→strlines "\n")
```

```
(string→strlines "\n\n")
```

Internally, `string`→`strlines` will need to convert its argument into a list of characters using `string`→`list`. We'll apply a helper function, `list`→`lines`, to that list.

What should `list`→`lines` produce?

One natural solution is a list, where each item is a list of characters that make up one line in `string`→`strlines`'s result.

:: (list→lines loc) turns loc into list of char lists based on newlines

:: list→lines: (listof Char) → (listof (listof Char))

:: Example:

```
(check-expect (list→lines '(#\a #\b #\newline #\c #\d))  
               '((#\a #\b) (#\c #\d)))
```

Thus `string`→`strlines` can be implemented as

```
:: string→strlines: Str → (listof Str)
```

```
(define (string→strlines str)
```

```
  (local
```

```
    ;; list→lines: (listof Char) → (listof (listof Char))
```

```
    [(define (list→lines loc)
```

```
      ... )]
```

```
  (map list→string
```

```
    (list→lines (string→list str)))))
```

We can use structural recursion on lists to write `list→lines`, but as we will see, the resulting code is awkward.

Rewriting the usual list template just a bit so we can refer to the recursive application several times without recomputing it gives us:

```
(define (list→lines loc)
  (cond [(empty? loc) ...]
        [else
         (local [(define r (list→lines (rest loc)))]
           (... (first loc) ... r ...))]))
```

If `loc` is nonempty and has `#\newline` as its first character, we `cons` an empty line onto the recursive result `r`.

If `loc` has, say, `#\a` as its first character, then this should be put at the front of the first line in the recursive result `r`.

But what if `r` is empty and has no first line?


```

(define (list→lines loc)
  (cond [(empty? loc) empty]
        [else (local [(define r (list→lines (rest loc)))]
                  (cond [(char=? (first loc) #\newline)
                         (cons empty r)]
                        [(empty? r)
                         (list (list (first loc)))]
                        [else (cons (cons (first loc) (first r))
                                    (rest r))]))]))

```

This approach works, but is pretty low-level and hard to reason about. If we reformulate the problem at a higher level, it turns out to be easier to reason about – but requires generative recursion to implement.

Instead of viewing the data as a sequence of characters, what if we view it as a sequence of lines?

```
(define (list→lines loc)
  (cond [(empty? loc) ...]
        [else (... (first-line loc) ...
                     (list→lines (rest-of-lines loc)) ... )]))
```

It's easy to fill in the “sequence of lines template”.

`:: list→lines: (listof Char) → (listof (listof Char))`

`(define (list→lines loc)`

`(cond [(empty? loc) empty]`

`[else (cons (first-line loc)`

`(list→lines (rest-of-lines loc))))])`

The two helper functions are also simpler, using structural recursion on lists.

:: (first-line loc) produces longest newline-free prefix of loc

:: first-line: (listof Char) \rightarrow (listof Char)

:: Examples:

(check-expect (first-line empty) empty)

(check-expect (first-line '#\newline) empty)

(check-expect (first-line '#\a #\newline) '#\a))

(define (first-line loc)

(cond [(empty? loc) empty]

[(char=? (first loc) #\newline) empty]

[else (cons (first loc) (first-line (rest loc))))])

:: (rest-of-lines loc) produces loc with everything

:: up to and including the first newline removed

:: rest-of-lines: (listof Char) \rightarrow (listof Char)

:: Examples:

(check-expect (rest-of-lines empty) empty)

(check-expect (rest-of-lines '(#\newline)) empty)

(check-expect (rest-of-lines '(#\a #\newline)) empty)

(check-expect (rest-of-lines '(#\a #\newline #\b)) '#\b))

```
(define (rest-of-lines loc)
  (cond [(empty? loc) empty]
        [(char=? (first loc) #\newline) (rest loc)]
        [else (rest-of-lines (rest loc))]))
```

Generalizing string→strlines

We can generalize `string→strlines` to accept a predicate that specifies when to break the string. We will call the pieces **tokens** instead of lines.

```
:: string→tokenstrs: (Char → Bool) Str → (listof Str)
```

```
(define (string→tokenstrs break? str)
```

```
  (map list→string
```

```
    (list→tokens break? (string→list str))))
```

```
(check-expect
```

```
  (string→tokenstrs char-whitespace? "One two\nthree")
```

```
  (list "One" "two" "three"))
```

:: string→tokenstrs: (Char → Bool) Str → (listof Str)

(define (string→tokenstrs break? str)

(local [(define (first-token loc)

(cond [(empty? loc) empty]

[(break? (first loc)) empty]

[else (cons (first loc) (first-token (rest loc))))]))

(define (remove-token loc)

(cond [(empty? loc) empty]

[(break? (first loc)) (rest loc)]

[else (remove-token (rest loc))]))


```

;; list→tokens: (listof Char) → (listof (listof Char))
(define (list→tokens loc)
  (cond [(empty? loc) empty]
        [else (cons (first-token loc)
                      (list→tokens (remove-token loc))))])
] ; end of local
(map list→string
     (list→tokens (string→list str))))

```

As an example, consider breaking a string up into “words” separated by **whitespace** (spaces, newlines).

In this case, we can use the built-in predicate `char-whitespace?`.

`(string→tokenstrs char-whitespace? "This is\na test.")`

`⇒ '("This" "is" "a" "test.")`

Tokenizing is especially important in interpreting or **parsing** a computer program or other specification. For instance,

```
(define (square x)  
  (* x x))
```

can be broken into twelve **tokens**.

Here, our simple method of using a character predicate will not work.

In later courses, we will discuss tokenizing in order to deal with situations such as a string representing a computer program, a Web page, or some other form of structured data.

More higher-order functions

Because generative recursion is so general, it is difficult to abstract it into one function.

However, we can create higher-order functions for certain patterns.

One example is accumulative recursion, which combines structural recursion with the updating of accumulators.

We need to be a little more specific: structural recursion on a list with one accumulator.

:: code from lecture module 9

```
(define (sum-list lon)
  (local [(define (sum-list/acc lst sofar)
            (cond [(empty? lst) sofar]
                  [else (sum-list/acc (rest lst)
                                       (+ (first lst) sofar))]))]
    (sum-list/acc lon 0)))
```

:: code from lecture module 7 rewritten to use **local**

```
(define (my-reverse lst0)
  (local [(define (my-rev/acc lst acc)
    (cond [(empty? lst) acc]
      [else (my-rev/acc (rest lst)
                          (cons (first lst) acc))]))])
  (my-rev/acc lst0 empty)))
```

The differences between these two functions are:

- the initial value of the accumulator;
- the computation of the new value of the accumulator, given the old value of the accumulator and the first element of the list.

```
(define (my-foldl combine base lst0)
  (local [(define (foldl-acc lst acc)
            (cond [(empty? lst) acc]
                  [else (foldl-acc (rest lst)
                                    (combine (first lst) acc))])])
    (foldl-acc lst0 base)))
```

```
(define (sum-list lon) (my-foldl + 0 lon))
```

```
(define (my-reverse lst) (my-foldl cons empty lst))
```


We noted earlier that 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) . . .)))`

What is the intuitive effect of the following application of `foldl`?

`(foldl f b (list x1 . . . xn-1 xn))`

The function `foldl` is provided in Intermediate Student.

What is the contract of `foldl`?

Goals of this module

You should understand the idea of generative recursion, why termination is not assured, and how a quantitative notion of a measure of progress towards a solution can be used to justify that such a function will return a result.

You should understand the examples given, particularly quicksort, line breaking, and tokenizing using a general break predicate.

You should be comfortable with accumulative recursion, and able to write accumulatively recursive helper functions with associated wrapper functions to hide the accumulator(s) from the user.

Graphs

Readings: Section 28

Backtracking algorithms

Backtracking algorithms try to find a route from an origin to a destination.

If the initial attempt does not work, such an algorithm “backtracks” and tries another choice.

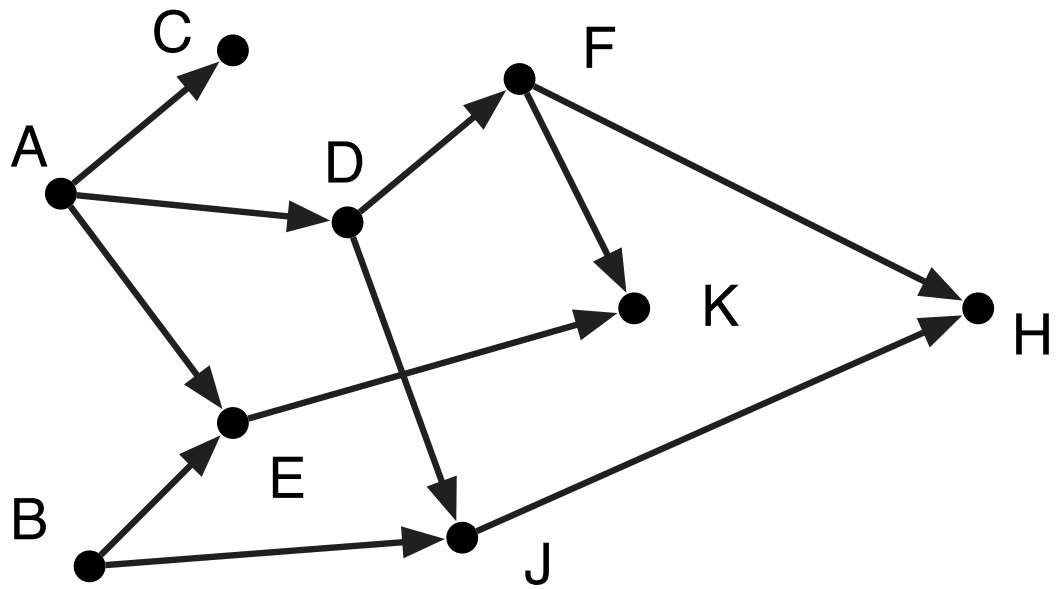
Eventually, either a route is found, or all possibilities are exhausted, meaning there is no route.

First, we look at a backtracking algorithm that works on an abstract representation of a map.

Directed graphs

A directed graph consists of a collection of **nodes** together with a collection of **edges**.

An edge is an ordered pair of nodes, which we can represent by an arrow from one node to another.



We have seen such graphs before.

Evolution trees and expression trees were both directed graphs of a special type.

An edge represented a parent-child relationship.

Graphs are a general data structure that can model many situations.

Computations on graphs form an important part of the computer science toolkit.

Graph terminology

Given an edge (v, w) , we say that w is an **out-neighbour** of v , and v is an **in-neighbour** of w .

A sequence of nodes v_1, v_2, \dots, v_k is a route or **path** of length $k - 1$ if $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$ are all edges.

If $v_1 = v_k$, this is called a **cycle**.

Directed graphs without cycles are called **DAGs** (directed acyclic graphs).

Representing graphs

We can represent a node by a symbol (its name), and associate with each node a list of its out-neighbours.

This is called the **adjacency list** representation.

More specifically, a graph is a list of pairs, each pair consisting of a symbol (the node's name) and a list of symbols (the names of the node's out-neighbours).

This is very similar to a parent node with a list of children.

Our example as data

'((A (C D E))

(B (E J))

(C ())

(D (F J))

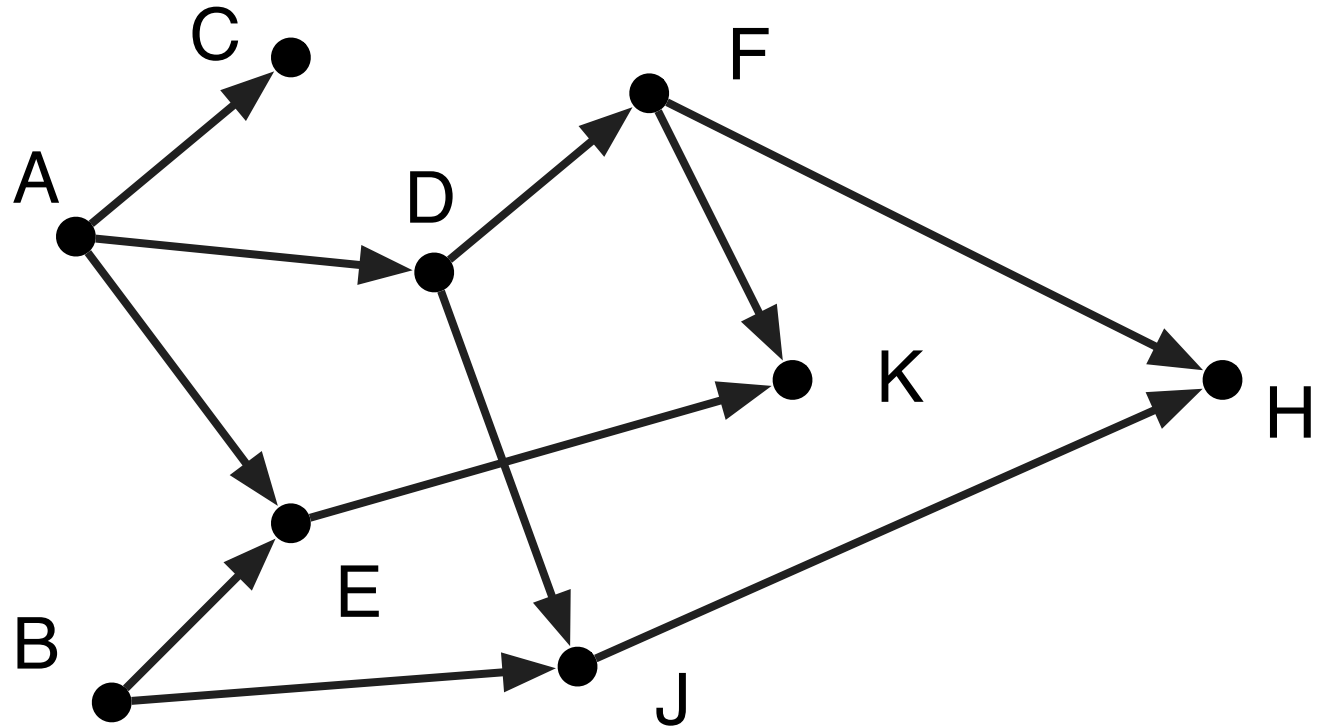
(E (K))

(F (K H))

(H ())

(J (H))

(K ()))



Data definitions

To make our contracts more descriptive, we will define a **Node** and a **Graph** as follows:

:: A Node is a Sym

:: A Graph is a (listof (list Node (listof Node)))

Structural recursion on graphs

We may be able to use structural recursion for some computations on graphs.

Since a graph is a list, we can use the list template.

The elements on this list are themselves lists of fixed length (two).

We can thus alter the structure template.

Instead of selector functions on a two-element structure, we use **first** and **second**.

The template for graphs

:: my-graph-fn: Graph \rightarrow Any

```
(define (my-graph-fn G)
```

```
  (cond [(empty? G) ...]
```

```
        [(cons? G) (... (first (first G)) ... ; first node in graph list
```

```
                        (second (first G)) ... ; list of adjacent nodes
```

```
                        (my-graph-fn (rest G)) ... )]))
```

Finding routes

A path in a graph can be represented by the list of nodes on the path.

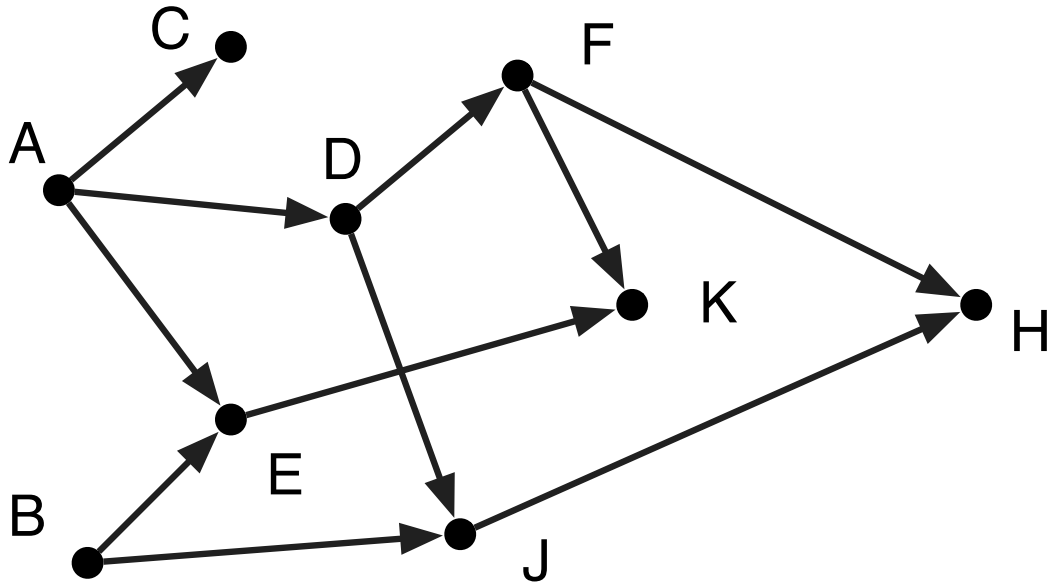
We wish to design a function `find-route` which consumes a graph plus origin and destination nodes, and produces a path from the origin to the destination, or `false` if no such path exists.

First we create an auxiliary function `neighbours` which consumes a node and a graph and produces the list of out-neighbours of the node.

Neighbours in our example

`(neighbours 'A G) ⇒ (list 'C 'D 'E)`

`(neighbours 'H G) ⇒ empty`



:: (neighbours v G) produces list of neighbours of v in G

:: neighbours: Node Graph \rightarrow (listof Node)

```
(define (neighbours v G)
```

```
  (cond [(empty? G) (error "vertex not in graph")]
```

```
        [(symbol=? v (first (first G))) (second (first G))]
```

```
        [else (neighbours v (rest G))]))
```


Cases for find-route

Structural recursion does not work for find-route; we must use generative recursion.

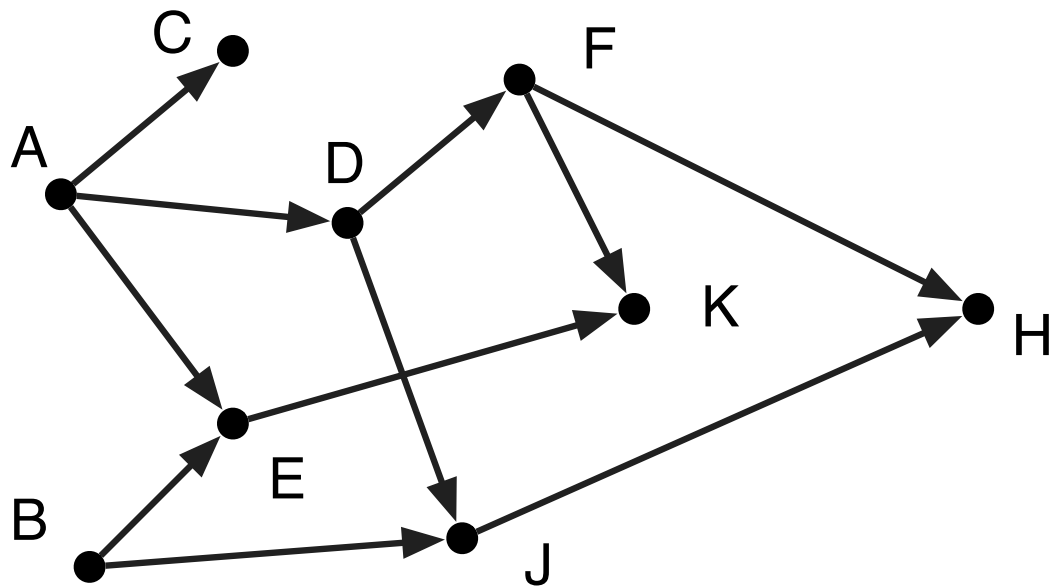
If the origin equals the destination, the path consists of just this node.

Otherwise, if there is a path, the second node on that path must be an out-neighbour of the origin node.

Each out-neighbour defines a subproblem (finding a route from it to the destination).

In our example, any route from A to H must pass through C, D, or E.

If we knew a route from C to H, or from D to H, or from E to H, we could create one from A to H.



This technique is called “backtracking” because the search for a route from C to H can be seen as moving forward in the graph looking for H.

If this search fails, the algorithm “backs up” to A and tries the next neighbour, D.

If we find a path from D to H, we can just add A to the beginning of this path.

We need to apply `find-route` on each of the out-neighbours of a given node.

All those out-neighbours are collected into a list associated with that node.

This suggests writing `find-route/list` which does this for the entire list of out-neighbours.

The function `find-route/list` will apply `find-route` to each of the nodes on that list until it finds a route to the destination.

This is the same recursive pattern that we saw in the processing of expression trees (and descendant family trees, in HtDP).

For expression trees, we had two mutually recursive functions, `eval` and `apply`.

Here, we have two mutually recursive functions, `find-route` and `find-route/list`.

:: (find-route orig dest G) finds route from orig to dest in G if it exists

:: find-route: Node Node Graph \rightarrow (anyof (listof Node) false)

```
(define (find-route orig dest G)
  (cond [(symbol=? orig dest) (list orig)]
        [else (local [(define nbrs (neighbours orig G))
                        (define route (find-route/list nbrs dest G))]
                  (cond [(false? route) route]
                        [else (cons orig route)]))]))
```

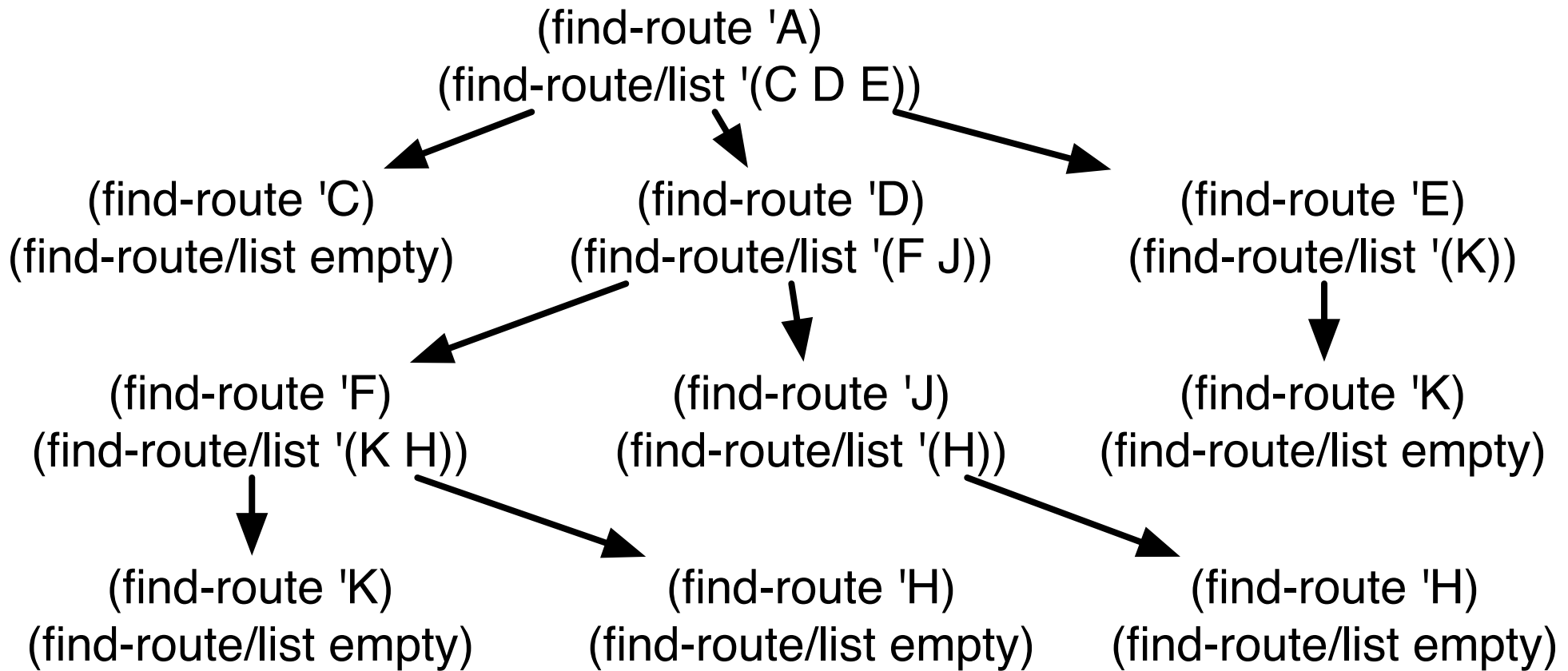
:: (find-route/list los dest G) produces route from
:: an element of los to dest in G, if one exists
:: find-route/list: (listof Node) Node Graph \rightarrow (anyof (listof Node) false)

```
(define (find-route/list los dest G)
  (cond [(empty? los) false]
        [else (local [(define route (find-route (first los) dest G))]
                  (cond [(false? route)
                        (find-route/list (rest los) dest G)]
                        [else route]))]))
```

We have the same problem doing a trace as with expression trees. A trace is a linear list of rewriting steps, but the computation is better visualized as a tree.

We will use an alternate visualization of the potential computation (which could be shortened if a route is found).

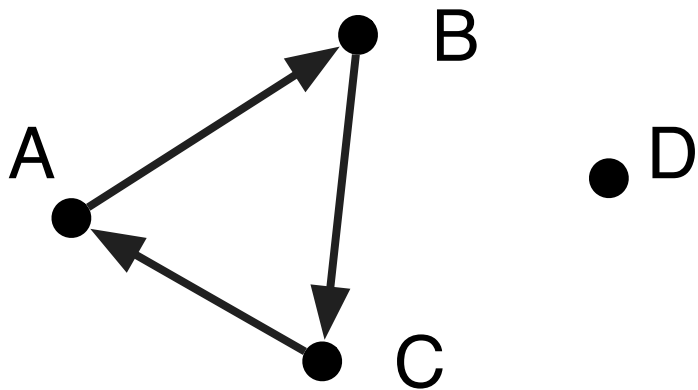
The next slide contains the trace tree. We have omitted the arguments `dest` and `G` which never change.

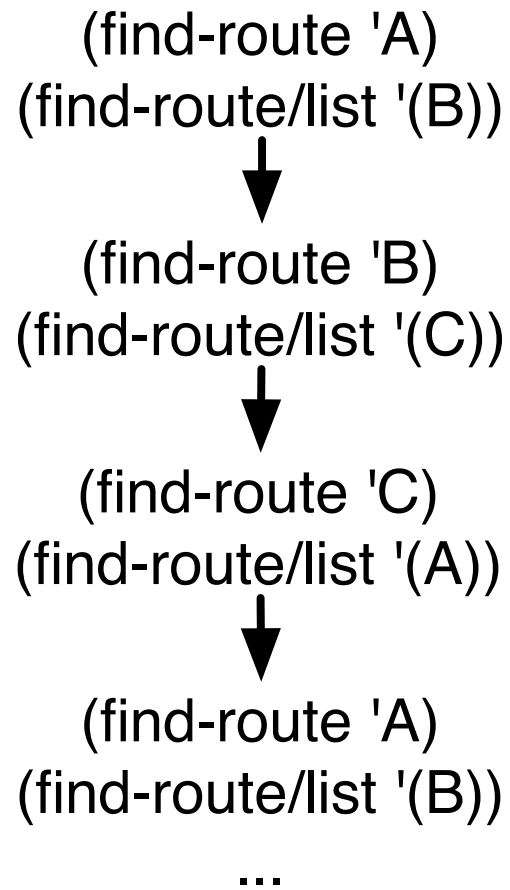


Termination of find-route

find-route may not terminate if there is a cycle in the graph.

Consider the graph `'((A (B)) (B (C)) (C (A)) (D ()))`. What if we try to find a route from A to D?





Termination for DAGs

In a directed acyclic graph, any route with a given origin is of finite length, and there are only finitely many of them.

Consider the evaluation of `(find-route orig dest G)`.

The number of recursive applications of `find-route` that occur during this evaluation is bounded above by the number of routes (any destination) originating from `orig`.

Thus `find-route` always terminates for directed acyclic graphs.

Backtracking in implicit graphs

The only place where real computation is done on the graph is in the `neighbours` function.

Backtracking can be used without having the entire graph available.

Example: nodes represent configurations of a board game (e.g. peg solitaire), edges represent legal moves.

The graph is acyclic if no configuration can occur twice in a game.

In another example, nodes could represent partial solutions of some problem (e.g. a sudoku puzzle, or the puzzle of putting eight mutually nonattacking queens on a chessboard).

Edges represent ways in which one additional element can be added to a solution.

The graph is naturally acyclic, since a new element is added with every edge.

The find-route functions for implicit backtracking look very similar to those we have developed.

The neighbours function must now generate the set of neighbours of a node based on some description of that node (e.g. the placement of pieces in a game).

This allows backtracking in situations where it would be inefficient to generate and store the entire graph as data.

Backtracking forms the basis of many artificial intelligence programs, though they generally add heuristics to determine which neighbour to explore first, or which ones to skip because they appear unpromising.

It is not hard to demonstrate situations where backtracking is highly inefficient.

Improving **find-route**

Recall that our backtracking function **find-route** would not work on graphs with cycles.

We can use accumulative recursion to solve this problem.

In the textbook, this is only done for a restricted form of graph; here we present the full solution.

First, we review the code for directed acyclic graphs.

:: (find-route orig dest G) finds route from orig to dest in G if it exists

:: find-route: Node Node Graph \rightarrow (anyof (listof Node) false)

```
(define (find-route orig dest G)
  (cond [(symbol=? orig dest) (list orig)]
        [else (local [(define nbrs (neighbours orig G))
                        (define route (find-route/list nbrs dest G))]
                    (cond [(false? route) route]
                          [else (cons orig route)]))]))
```

:: (find-route/list los dest G) produces route from
:: an element of los to dest in G, if one exists
:: find-route/list: (listof Node) Node Graph \rightarrow (anyof (listof Node) false)
(define (find-route/list los dest G)
 (cond [(empty? los) false]
 [else (local [(define route (find-route (first los) dest G))]
 (cond [(false? route) (find-route/list (rest los) dest G)]
 [else route]))]))

To make backtracking work in the presence of cycles, we need a way of remembering what nodes have been visited (along a given path).

Our accumulator will be a list of visited nodes.

We must avoid visiting a node twice.

The simplest way to do this is to add a check in [find-route/list](#).

:: find-route/list: (listof Node) Node Graph (listof Node)

:: \rightarrow (anyof (listof Node) false)

```
(define (find-route/list los dest G visited)
  (cond [(empty? los) false]
        [(member? (first los) visited)
         (find-route/list (rest los) dest G visited)]
        [else (local [(define route (find-route/acc (first los)
                                                       dest G visited))]
                  (cond [(false? route)
                         (find-route/list (rest los) dest G visited)]
                        [else route]))]))
```

The code for `find-route/list` does not add anything to the accumulator (though it uses the accumulator).

Adding to the accumulator is done in `find-route/acc` which applies `find-route/list` to the list of neighbours of some origin node.

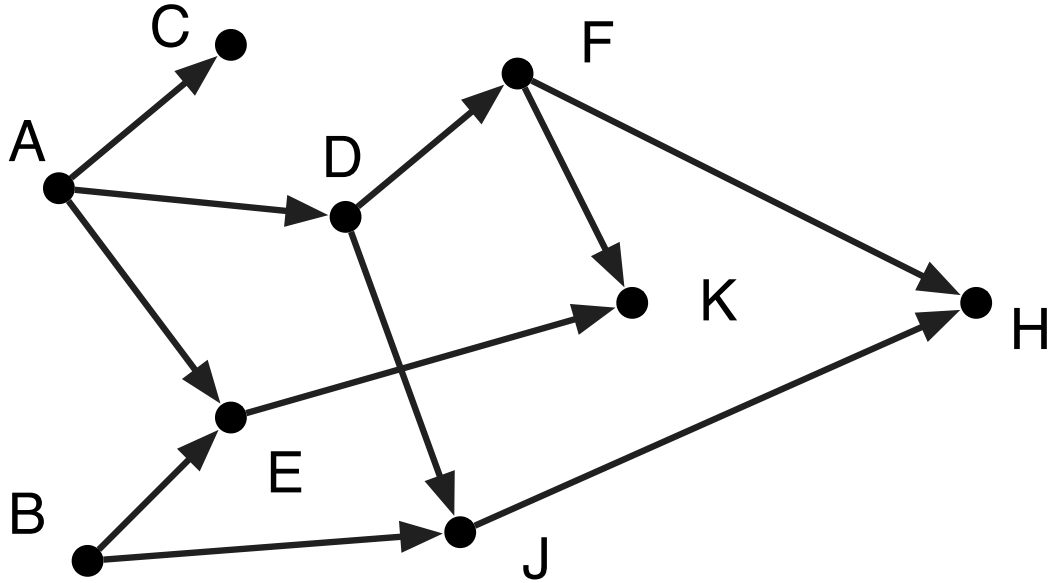
That origin node must be added to the accumulator passed as an argument to `find-route/list`.

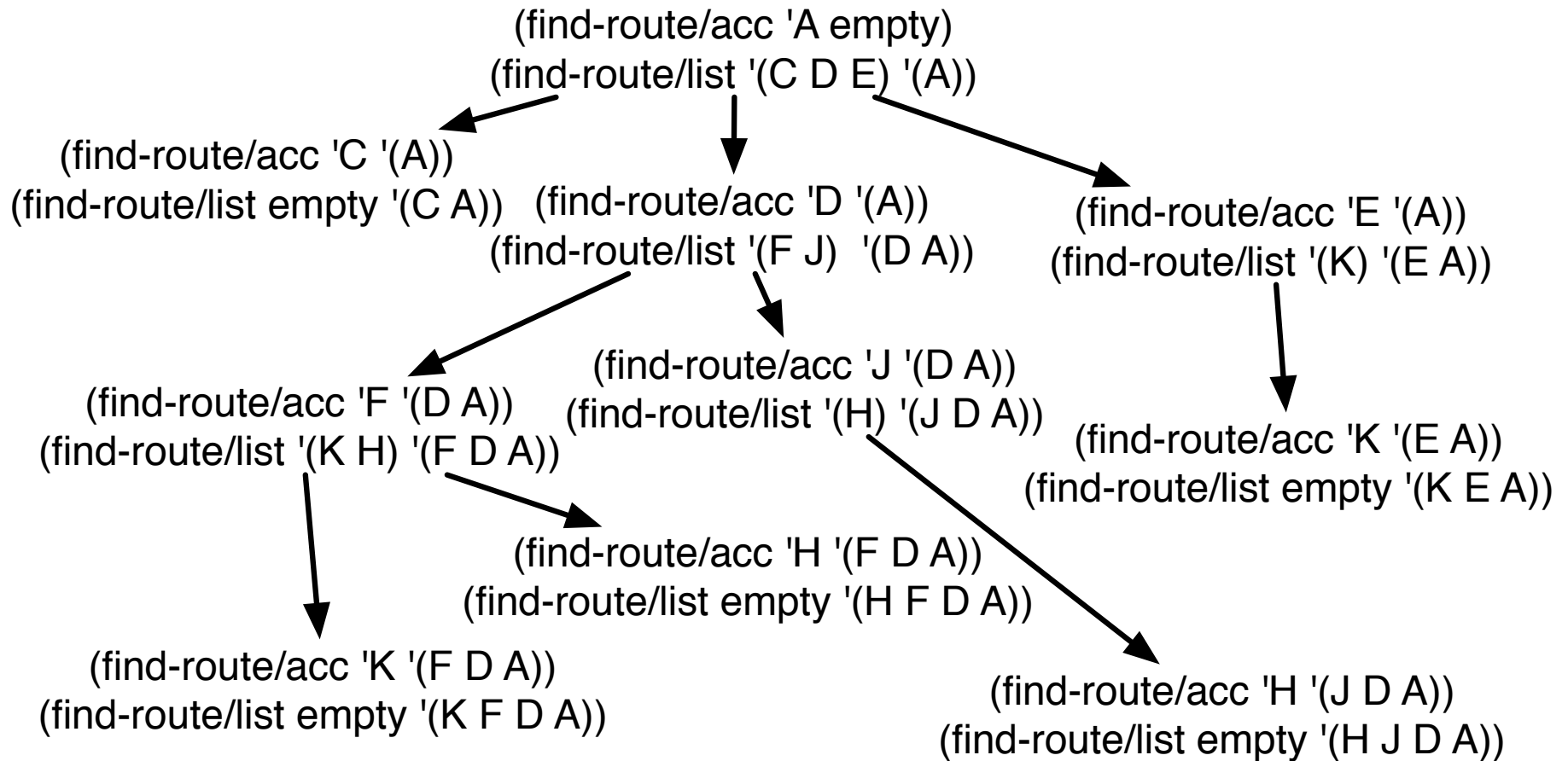
:: find-route/acc: Node Node Graph (listof Node)

:: \rightarrow (anyof (listof Node) false)

```
(define (find-route/acc orig dest G visited)
  (cond [(symbol=? orig dest) (list orig)]
        [else (local [(define nbrs (neighbours orig G))
                        (define route (find-route/list nbrs dest G
                                                         (cons orig visited)))]
                  (cond [(false? route) route]
                        [else (cons orig route)]))]))
```

Revisiting our example



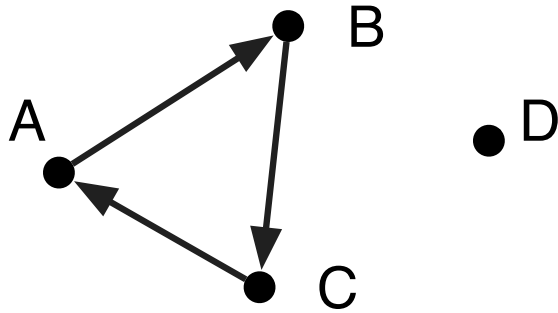


Note that the value of the accumulator in `find-route/list` is always the reverse of the path from A to the current origin (first argument).

This example has no cycles, so the trace only convinces us that we haven't broken the function on acyclic graphs, and shows us how the accumulator is working.

But it also works on graphs with cycles.

The accumulator ensures that the depth of recursion is no greater than the number of nodes in the graph, so `find-route` terminates.



(find-route/acc 'A empty)
(find-route-list '(B) '(A))



(find-route/acc 'B '(A))
(find-route-list '(C) '(B A))

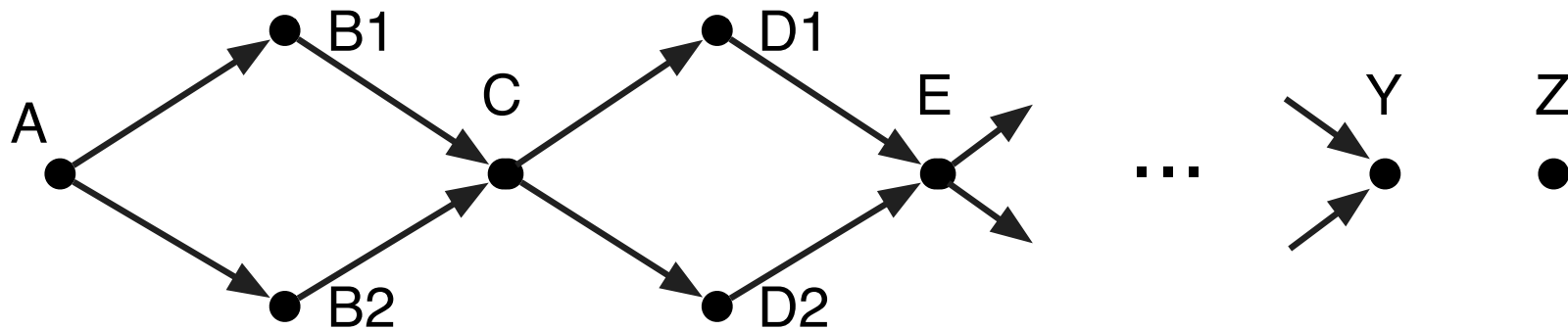


(find-route/acc 'C '(B A))
(find-route-list '(A) '(C B A))
no further recursive calls

In practice, we would write a wrapper function for users which would avoid their having to specify the initial value of the accumulator, as we have with the other examples in this module.

Backtracking now works on graphs with cycles, but it can be inefficient, even if the graph has no cycles.

If there is no path from the origin to the destination, then `find-route` will explore every path from the origin, and there could be an exponential number of them.



If there are d diamonds, then there are $3d + 2$ nodes in the graph, but 2^d paths from A to Y, all of which will be explored.

Making **find-route/acc** efficient

Applying **find-route/acc** to origin A results in **find-route/list** being applied to '(B1 B2), and then **find-route/acc** being applied to origin B1.

There is no route from B1 to Z, so this will produce **false**, but in the process, it will visit all the other nodes of the graph except B2 and Z.

find-route/list will then apply **find-route/acc** to B2, which will visit all the same nodes.

When `find-route/list` is applied to the list of nodes `los`, it first applies `find-route/acc` to `(first los)` and then, if that fails, it applies itself to `(rest los)`.

To avoid revisiting nodes, the failed computation should pass the list of nodes it has seen on to the next computation.

It will do this by returning the list of visited nodes instead of `false` as the sentinel value. However, we must be able to distinguish this list from a successfully found route (also a list of nodes), so we make a new *type* to use as the sentinel:

```
(define-struct noroute (visited))
```

:: find-route/list: ... \rightarrow (anyof (listof Node) NoRoute)

```
(define (find-route/list los dest G visited)
  (cond [(empty? los) (make-noroute visited)]
        [(member? (first los) visited)
         (find-route/list (rest los) dest G visited)]
        [else (local [(define route (find-route/acc (first los)
                                                       dest G visited))]
                  (cond [(noroute? route)
                         (find-route/list (rest los) dest G
                                           (noroute-visited route))]
                        [else route]))]))
```


:: find-route/acc: Node Node Graph (listof Node)

:: \rightarrow (anyof (listof Node) NoRoute)

```
(define (find-route/acc orig dest G visited)
  (cond [(symbol=? orig dest) (list orig)]
        [else (local [(define nbrs (neighbours orig G))
                        (define route (find-route/list nbrs dest G
                                                         (cons orig visited)))]
                  (cond [(noroute? route) route]
                        [else (cons orig route)]))]))
```

;; find-route: Node Node Graph \rightarrow (anyof (listof Node) false)

(**define** (find-route orig dest G)

(**local** [(**define** route (find-route/acc orig dest G empty))]

(**cond** [(noroute? route) false]

[**else** route]))))

With these changes, `find-route` runs much faster on the diamond graph.

How efficient is our final version of `find-route`?

Each node is added to the **visited** accumulator at most once, and once it has been added, it is not visited again.

Thus **find-route/acc** is applied at most n times, where n is the number of nodes in the graph.

One application of **find-route/acc** (not counting recursions) takes time roughly proportional to n (mostly in **neighbours**).

The total work done by **find-route/acc** is roughly proportional to n^2 .

One application of **find-route/list** takes time roughly proportional to n (mostly in **member?**).

find-route/list is applied at most once for every node in a neighbour list, that is, at most m times, where m is the number of edges in the graph.

The total cost is roughly proportional to $n(n + m)$.

We will see how to make **find-route** even more efficient in later courses, and see how to formalize our analyses.

Knowledge of efficient algorithms, and the data structures that they utilize, is an essential part of being able to deal with large amounts of real-world data.

These topics are studied in CS 240 and CS 341 (for majors) and CS 234 (for non-majors).

Goals of this module

You should understand directed graphs and their representation in Racket.

You should be able to write functions which consume graphs and compute desired values.

You should understand and be able to implement backtracking on explicit and implicit graphs.

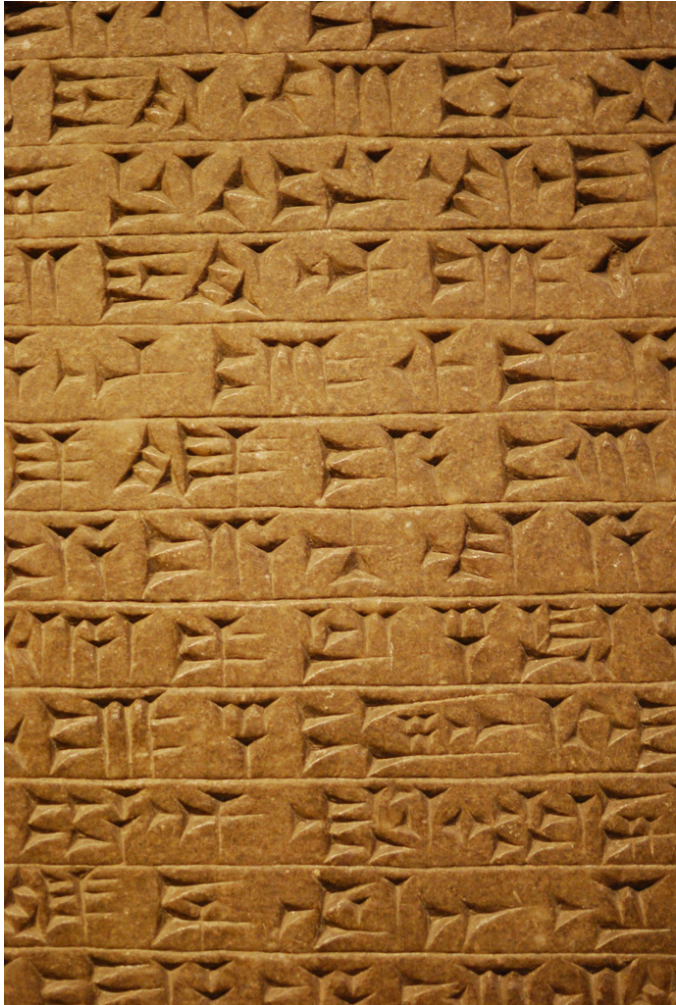
You should understand the performance differences in the various versions of `find-route` and be able to write more efficient functions using appropriate recursive techniques.

History

This is how one pictures the angel of history. His face is turned toward the past. Where we perceive a chain of events, he sees one single catastrophe which keeps piling wreckage and hurls it in front of his feet. The angel would like to stay, awaken the dead, and make whole what has been smashed. But a storm is blowing in from Paradise; it has got caught in his wings with such a violence that the angel can no longer close them. The storm irresistibly propels him into the future to which his back is turned, while the pile of debris before him grows skyward. This storm is what we call progress.

(Walter Benjamin, 1940)

The Dawn of Computation



Babylonian
cuneiform
circa 2000 B.C.
(Photo by Matt Neale)

Early Computation

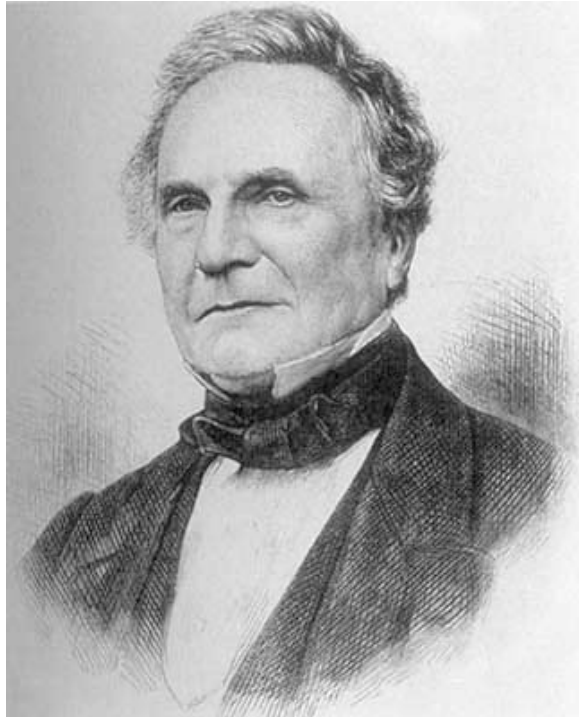
“computer” = human being performing computation

Euclid’s algorithm circa 300 B.C.

Abu Ja’far Muhammad ibn Musa Al-Khwarizmi’s books on algebra and arithmetic computation using Indo-Arabic numerals, circa 800 A.D.

Isaac Newton (1643-1727)

Charles Babbage (1791-1871)



Difference Engine (1819)

Analytical Engine (1834)

Mechanical computation for
military applications

The specification of
computational operations
was separated from their
execution

Babbage's designs were
technically too ambitious

Ada Augusta Byron (1815-1852)



Assisted Babbage in explaining and promoting his ideas

Wrote articles describing the operation and use of the Analytical Engine

The first computer scientist?

David Hilbert (1862-1943)

Formalized the axiomatic treatment of
Euclidean geometry

Hilbert's 23 problems (ICM, 1900)

Problem #2: Is mathematics consistent?

The meaning of proof

Axiom: $\forall n : n + 0 = n.$

Math statement: “The square of any even number is even.”

Formula: $\forall n (\exists k : n = k + k \Rightarrow \exists m : m + m = n * n)$

Proof: Finite sequence of axioms (basic true statements) and derivations of new true statements (e.g. ϕ and $\phi \rightarrow \sigma$ yield σ).

Theorem: A mathematical statement ϕ together with a proof deriving ϕ within a given system of axioms and derivation rules.

Hilbert's questions (1920's)

Is mathematics complete? Meaning: for any formula ϕ , if ϕ is true, then ϕ is provable.

Is mathematics consistent? Meaning: for any formula ϕ , there aren't proofs of both ϕ and $\neg\phi$.

Is there a procedure to, given a formula ϕ , produce a proof of ϕ , or show there isn't one?

Hilbert believed the answers would be “yes”.

Kurt Gödel (1906-78)

Gödel's answers to Hilbert (1929-30):

Any axiom system powerful enough to describe arithmetic on integers is not complete.

If it is consistent, its consistency cannot be proved within the system.

Sketch of Gödel's proof

Define a mapping between logical formulas and numbers.

Use it to define mathematical statements saying “This number represents a valid formula”, “This number represents a sequence of valid formulae”, “This number represents a valid proof”, “This number represents a provable formula”.

Construct a formula ϕ represented by a number n that says “The formula represented by n is not provable”. The formula ϕ cannot be false, so it must be true but not provable.

What remained of Hilbert's questions

Is there a procedure which, given a formula ϕ , either proves ϕ , shows it false, or correctly concludes ϕ is not provable?

The answer to this requires a precise definition of “a procedure”, in other words, a formal model of computation.

Alonzo Church (1903-1995)

Set out to give a final “no” answer to this last question

With his student Kleene, created notation to describe functions on the natural numbers.

Church and Kleene's notation

They wanted to modify Russell and Whitehead's notation for the class of all x satisfying a predicate f : $\hat{x} f(x)$.

But their notion was somewhat different, so they tried putting the caret before: $\hat{} x$.

Their typewriter could not type this, but had Greek letters.

Perhaps a capital lambda? Λx .

Too much like the symbol for logical AND: \wedge .

Perhaps a lower-case lambda? λx .

The lambda calculus

The function that added 2 to its argument would be represented by $\lambda x.x + 2$.

The function that subtracted its second argument from its first would be written $\lambda x.\lambda y.x - y$.

fx applies function f to argument x .

fxy means $(fx)y$ (left-associativity).

To prove something is impossible to express in some notation, the notation should be as simple as possible.

To make things even simpler, the lambda calculus did not permit naming of functions (only parameters), naming of constants like 2, or naming of functions like +.

It had three grammar rules and one reduction rule (function application).

How could it say anything at all?

Numbers from nothing (Cantor-style)

$0 \equiv \emptyset$ or $\{\}$ (the empty set)

$1 \equiv \{\emptyset\}$

$2 \equiv \{\{\emptyset\}, \emptyset\}$

In general, n is represented by the set containing the sets representing $n - 1, n - 2, \dots, 0$.

This is the way that arithmetic can be built up from the basic axioms of set theory.

Numbers from nothing (Church-style)

$0 \equiv \lambda f. \lambda x. x$ (the function which ignores its argument and returns the identity function)

$1 \equiv \lambda f. \lambda x. f x$ (the function which, when given as argument a function f , returns the same function).

$2 \equiv \lambda f. \lambda x. f(f x)$ (the function which, when given as argument a function f , returns f composed with itself or $f \circ f$).

In general, n is the function which does n -fold composition.

With some care, one can write down short expressions for the addition and multiplication functions.

Similar ideas will create Boolean values, logical functions, and conditional expressions.

General recursion without naming is harder, but still possible.

The lambda calculus is a general model of computation.

Church's proof

Church proved that there was no computational procedure to tell if two lambda expressions were equivalent (represented the same function).

His proof mirrored Gödel's, using a way of encoding lambda expressions using numbers, and provided a “no” answer to the idea of deciding provability of formulae.

This was published in 1936.

Independently, a few months later, a British mathematician came up with a simpler proof.

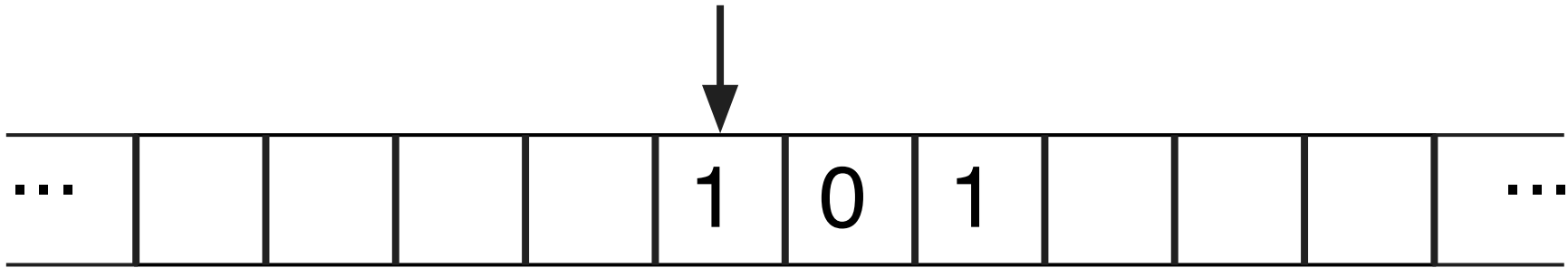
Alan Turing (1912-1954)

Turing defined a different model of computation, and chose a different problem to prove uncomputable.

This resulted in a simpler and more influential proof.

Turing's model of computation

f: state char \rightarrow state char move



Finite state control plus unbounded storage tape

Turing's proof (1936)

Turing showed how to code the finite state control using characters.

He then assumed that there was a machine that could process such a description and tell whether the coded machine would halt or not when fed its own description as an input.

Using this machine, one can define a second machine that acts on this information.

The second machine uses the first machine to see if its input represents a coded machine which halts when fed its own description.

If so, the second machine runs forever; otherwise, it halts.

Feeding the description of the second machine to itself creates a contradiction: it halts iff it doesn't halt.

So the first machine cannot exist.

Turing's proof also demonstrates the undecidability of proving formulae.

Advantages of Turing's ideas

Turing's ideas can be adapted to give a similar proof in the lambda calculus model.

Upon learning of Church's work, Turing quickly sketched the equivalence of the two models.

Turing's model bears a closer resemblance to an intuitive idea of real computation.

It would influence the future development of hardware and thus software, even though reasoning about programs is more difficult in it.

Turing went to America to study with Church at Princeton, earning his PhD in 1939.

During the war, he was instrumental in an effort to break encrypted German radio traffic, resulting in the development of what we now know to be the world's first working electronic computer (Colossus).

Turing made further contributions to hardware and software design in the UK, and to the field of artificial intelligence, before his untimely death in 1954.

John von Neumann (1903-1957)

von Neumann was a founding member of the Institute for Advanced Study at Princeton.

In 1946 he visited the developers of ENIAC at the University of Pennsylvania, and wrote an influential “Report on the EDVAC” regarding its successor.

Features: random-access memory, CPU, fetch-execute loop, stored program.

Lacking: support for recursion (unlike Turing’s UK designs)

Grace Murray Hopper (1906-1992)

Wrote first compiler, defined first English-like data processing language

Ideas later folded into COBOL (1959)



FORTRAN (1957)

Early programming language influenced by architecture.

```
      INTEGER FN, FNM1, TEMP
      FN = 1
      FNM1 = 0
      DO 20 I = 1, 10, 1
      PRINT 10, I, FN
10    FORMAT(I3, 1X, I3)
      TEMP = FN + FNM1
      FNM1 = FN
20    FN = TEMP
```

FORTRAN was designed by John Backus, and became the dominant language for numerical and scientific computation. Backus also invented a notation for language description that is popular in programming language design today.

Backus won the Turing Award in 1978, and used the associated lecture to criticize the continued dominance of von Neumann's architectural model and the programming languages inspired by it.

He proposed a functional programming language for parallel/distributed computation.

FORTRAN and COBOL, reflecting the Turing - von Neumann approach, dominated practical computing through most of the '60's and '70's.

Many other computer languages were defined, enjoyed brief and modest success, and then were forgotten.

Church's work proved useful in the field of operational semantics, which sought to treat the meaning of programs mathematically.

It also was inspirational in the design of a still-popular high-level programming language called Lisp.

John McCarthy (1927-2011)

McCarthy, an AI researcher at MIT, was frustrated by the inexpressiveness of machine languages and the primitive programming languages arising from them (no recursion, no conditional expressions).

In 1958, he designed and implemented Lisp (LISt Processor), taking ideas from the lambda calculus and the theory of recursive functions.

His 1960 paper on Lisp described the core of the language in terms that CS 135 students would recognize.

McCarthy's Lisp

McCarthy defined these primitive functions: `atom` (the negation of `cons?`), `eq`, `car` (`first`), `cdr` (`rest`), and `cons`.

He also defined the special forms `quote`, `lambda`, `cond`, and `label` (`define`).

Using these, he showed how to build many other useful functions.

The evolution of Lisp

The first implementation of Lisp, on the IBM 704, could fit two machine addresses (15 bits) into parts of one machine word (36 bits) called the address and decrement parts. Machine instructions facilitated such manipulation.

This led to the language terms `car` and `cdr` which persist in Racket and Lisp to this day.

Lisp quickly evolved to include proper numbers, input/output, and a more comprehensive set of built-in functions.

Lisp became the dominant language for artificial intelligence implementations.

It encouraged redefinition and customization of the language environments, leading to a proliferation of implementations.

It also challenged memory capabilities of 1970's computers, and some special-purpose “Lisp machines” were built.

Modern hardware is up to the task, and the major Lisp groups met and agreed on the Common Lisp standard in the 1980's.

The origins of Scheme

In 1976, researcher Carl Hewitt designed an ambitious Lisp-like language called Planner, with facilities for automated backtracking for goal search.

Researcher Gerald Sussman, with others, created Conniver, a version which made the control structures more explicit and more efficient.

Hewitt also proposed the *actors* theory of distributed object-oriented computation, with facilities to move computation from one “actor” to another.

Sussman and his graduate student Guy Steele were having trouble understanding some of the implications of the actors model, so they wrote a small Lisp interpreter for a “toy” actors language.

Sussman suggested using lexical scoping instead of the dynamic scoping used by Lisp, and a single namespace (Lisp has separate namespaces for functions and variables).

The migration of computation was captured by the idea of *continuations*, which made the unevaluated part of a computation into a manipulable object.

Sussman and Steele realized that function creation and actor creation were almost identical, as were function application, message passing, and actor invocation.

This simplified their language even further.

They wanted to call it Schemer, but there was a six-character file name limit on their system, so it became Scheme.

Research groups at other universities started using Scheme to study programming languages.

Sussman and Steele today

Steele went to work for Thinking Machines and then Sun Microsystems, where he became an early member of the team developing the Java programming language.

Sussman, together with colleague Hal Abelson, started using Scheme in the undergraduate program at MIT. Their textbook, “Structure and Interpretation of Computer Programs”, is used there, as well as at many other major institutions.

The authors of the HtDP textbook developed an extension of Scheme (PLT Scheme) and its learning environment (DrScheme) to remedy the following perceived deficiencies of SICP:

- lack of programming methodology
- complex domain knowledge required
- steep, frustrating learning curve
- insufficient preparation for future courses

As PLT Scheme and the teaching languages diverged further from Sussman and Steele's Scheme, they renamed their language Racket in 2010.

Summing up CS 135

With only a few language constructs (**define**, **cond**, **define-struct**, **cons**, **local**, **lambda**) we have described and implemented ideas from introductory computer science in a brief and natural manner.

We have done so without many of the features (static types, mutation, I/O) that courses using conventional languages have to introduce on the first day. The ideas we have covered carry over into languages in more widespread use.

We hope you have been convinced that the goal of computer science is to implement useful computation in a way that is correct and efficient as far as the machine is concerned, but that is understandable and extendable as far as other humans are concerned.

These themes will continue in CS 136, but new themes will be added, and a new programming language using a different paradigm (though we will continue to use Racket as well).

Looking ahead to CS 136

We have been fortunate to work with very small languages (the teaching languages) writing very small programs which operate on small amounts of data.

In CS 136, we will broaden our scope, moving towards the messy but also rewarding realm of the “real world”.

The main theme of CS 136 is scalability: what are the issues which arise when things get bigger, and how do we deal with them?

How do we organize a program that is bigger than a few screenfuls?

How do we reuse and share code, apart from cutting-and-pasting it into a new program file?

How do we design programs so that they run efficiently?

What changes might be necessary to our notion of types and to the way we handle errors when there is a much greater distance in time and space between when the program is written and when it is run?

When is it appropriate to abstract away from implementation details for the sake of the big picture, and when must we focus on exactly what is happening at lower levels for the sake of efficiency?

These are issues which arise not just for computer scientists, but for anyone making use of computation in a working environment.

We can build on what we have learned this term in order to meet these challenges with confidence.