Welcome to CS 135 (Fall 2019)

Instructors: Byron Weber Becker, Charles Clarke, Mark Giesbrecht, Dan Holtby, Kevin Lanctot, Adrian Reetz

Other course personnel: see website for details

- ISAs (Instructional Support Assistants)
- IAs (Instructional Apprentices)
- ISC (Instructional Support Coordinator)

Web page (main information source):
https://www.student.cs.uwaterloo.ca/~cs135/
Who Am I

• Kevin Lanctot, kevin.lanctot@uwaterloo.ca

• Office: DC 2131 (near the sky walk to MC and M3)

• Things to know about me:
  – I’m a talker not a typer.
  – I only check my email a few times a day.
  – Last name is pronounced long-k toe, i.e. “long toe” with the “k” sound after long.

• Note: lecture material, assignments, and exams are the same across all sections of CS135.
Additional Notes

- Besides the “official” course notes I will create some additional slides.

- They will be merged into the official slides but have “Additional Notes” and a slide number like 1.2 at the bottom.

- They will be available at the course web site
  https://www.student.cs.uwaterloo.ca/~cs135/cc/instructor_materials/

- Until the web site gets finalized, they’ll also be available at
  https://learn.uwaterloo.ca/
  under the Content tab of CS135 by 9 pm the evening before each lecture.
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)

The approach is: learn how to think about solving problems using a computer.
Themes of the course

• One of our goals is to develop good programming habits.

• May seem unimportant for small or simple programs but the complexity will eventually become overwhelming.

• e.g.

  https://informationisbeautiful.net/visualizations/million-lines-of-code/
Lectures

Tuesdays and Thursdays 8:30 am – 9:50 am.

Textbook: “How to Design Programs (First Edition)” (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi (find link on web site)

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

Participation marks: to encourage active learning
Participation marks: to encourage active learning

• Based on “clickers” (purchase at the Bookstore; register in A0) and several multiple-choice questions in each lecture

• One mark for any answer; second mark for correct answer

• We use the best 75% across the entire term to calculate 5% of the final mark.

• No sharing clickers: each clicker must be used by only one student in CS 135.

• No bringing your friend’s clicker to class, of course.

• You must attend the section you’re officially registered in in order to get credit for your participation.
Tutorials

• 16 sections on Fridays (starting tomorrow but attendance for this one is optional)

• Reinforces lectures with additional examples and problem-solving sessions

• Often directly applicable to the upcoming assignment

• Take your laptop and clicker

You should definitely be attending if your assignment marks are below 80%.
Assignments

Timing: About 10 assignments, typically due Tuesday at 9:00PM

Software: DrRacket v7.3 (http://racket-lang.org)

Computer labs: MC 3003, 3004, 3005, 3027, 2062, 2063. Available for your use, but no scheduled labs. Most students use their own computers.

A0: Due Tuesday, Sept 10. Must complete before you are allowed to submit any subsequent assignment

Exams

- Two midterms
  - Sept 30, 7:00–8:20 pm (80 minutes long)
  - Nov 4, 7:00–8:50 pm (110 minutes long)
- One final (date to be determined by the Registrar)

Do not make holiday travel plans before you know the date of all your final exams AND take into account the snow dates.
Marking scheme

• 20% Assignments (roughly weekly)
• 10% Midterm 1
• 15% Midterm 2
• 50% Final exam
• 5% Participation (on best 75% of the clicker questions)

To pass the course:

⇒ Your weighted assignment average must be 50% or greater.
⇒ Your weighted exam average must be 50% or greater.
Getting help

• Tutors have regular office hours. Schedule on web site.
• Instructors also have office hours.
• Piazza: An on-line forum where you can ask questions, other students and instructors answer them.
  – Regularly check the official assignment pinned posts
  – Use meaningful subject headings (not just “A3 problem”; what’s your specific problem?)
  – Search previous posts before posting; Don’t duplicate!
  – Possible to post privately if necessary
Suggestions for success

Read the CS135 Survival Guide as soon as possible. Find it on the course web site under “Help”.

- Keep up with your assignments. Start them early. **This is key!**
- Go over your assignments and exams; learn from your mistakes.
- Attend lectures; take notes
- Visit office hours as needed; earlier is better.
- Follow our advice on approaches to writing programs (e.g. design recipe, templates).
- Read your mail sent to your UW email account.
Suggestions for success (cont.)

- Keep up with the readings (keep ahead if possible).
- 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.
Academic integrity

• You must do your own work.

• Policy 71 - Student Discipline: plagiarism, sharing assignments, etc.

• Running out of time? It is better to hand in a partial assignment or nothing than to hand in someone else’s work.

• Be careful about posting code to Piazza. If it looks like it could have come from your assignment, don’t post it (publicly).
Academic Integrity

• Do your own work. ⇒ Do not copy another student’s work.
  – You *may talk* to other people about how to do an assignment, but *do not write or record anything*.
  – You *cannot view or share* another person’s code.

• Standard penalty for first offence is 0 on the assignment and -5% on the final grade.

• Suspension/expulsion for 2nd offence *in any course*.

• We have software that detects similar code.
Intellectual property

The teaching material used is 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.
Goals of this module

You should understand how the course is organized [1–7].

You should be familiar with the course resources available to you [9].

You should know what you need to do to earn the mark you desire [8,10,11].

You should know how to avoid plagiarism [12].
Module 02: Functions

Readings:

• HTDP, sections 1-3
• Survival and Style guides

Topics:

• Programming language design [1-4]
• The DrRacket environment [5-6]
• Values, expressions, & functions [7-21]
• Defining functions [22-30]
• Scope [31]
• Programming in DrRacket [32-34]
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
- graduated set of teaching languages are a subset of Racket
Functional vs. imperative

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.
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 (now) and Definitions (later)
- 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.
Values, expressions, & functions

Values are *numbers or other mathematical objects.*
Examples: $5$, $\frac{4}{9}$, $\pi$.

Expressions *combine values with operators and functions.*
Examples: $5 + 2$, $\sin(2\pi)$, $\frac{\sqrt{2}}{100\pi}$.

Functions *generalize similar expressions.*
Example...
Values, expressions, & functions (cont)

Values are numbers or other mathematical objects.

Expressions combine values with operators and functions.

Functions generalize similar expressions.

Example:

\[ 3^2 + 4(3) + 2 \]
\[ 6^2 + 4(6) + 2 \]
\[ 7^2 + 4(7) + 2 \]

are generalized by the function

\[ f(x) = x^2 + 4x + 2. \]
Functions in mathematics

Definitions: \( f(x) = x^2 \), \( g(x, y) = x + y \)

Function definitions in mathematics consist of three components:

1. the name of the function (e.g. \( g \))

2. its parameters (e.g. \( x, y \))

3. an algebraic expression using the parameters as placeholders for values to be supplied in the future
Function application

Definitions: \( f(x) = x^2,\ g(x, y) = x + y \)

An application of a function supplies arguments for the parameters, which are substituted into the algebraic expression.

An example: \( g(1, 3) = 1 + 3 = 4 \)

The arguments supplied may themselves be applications.

Example: \( g(g(1, 3), f(3)) \)
Function application (cont)

Definitions: \( f(x) = x^2 \), \( g(x, y) = x + y \)

We evaluate each of the arguments to yield values.

Evaluation by substitution:

\[
\begin{align*}
g(g(1, 3), f(3)) &= \\
g(1 + 3, f(3)) &= \\
g(4, f(3)) &= \\
g(4, 3^2) &= \\
g(4, 9) &= 4 + 9 = 13
\end{align*}
\]
Many possible substitutions

Definitions: \( f(x) = x^2, \ g(x, y) = x + y \)

There are many mathematically valid substitutions:

\[
g(g(1, 3), f(3)) = g(1 + 3, f(3))
\]

\[
g(g(1, 3), f(3)) = g(g(1, 3), 3^2)
\]

\[
g(g(1, 3), f(3)) = g(1, 3) + f(3)
\]

We’d like a **canonical form** (**a standard form**) for two reasons:

- Easier for us to think about
- When we extend this idea to programming, we’ll find cases where different orderings result in different values
Canonical substitutions

There are two rules for canonical substitutions

1. *Functions are applied to values*

2. When there is a choice of possible substitutions, *always take the leftmost choice.*

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, \(\text{we don't need parentheses to specify order of operations:}\)

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).
Infix vs. Prefix Notation

- **Infix notation** operators go between values: $1 + 2$.
  - need precedence rules to determine order of evaluation
  - BEDMAS: brackets, exponents, division/multiplication, addition/subtraction
  - if operators have the same precedence (e.g. addition/subtraction) then evaluation left to right

- **Prefix notation**: operators go before values: $(+ 1 2)$
Evaluating a Racket expression

*We use a process of substitution*, just as with our mathematical expressions.

Each step is indicated using the *yields symbol* $\Rightarrow$.

$$(\ast (\neg 6 4) (\div 3 2))$$

$\Rightarrow (\ast 2 (\div 3 2))$

$\Rightarrow (\ast 2 5)$

$\Rightarrow 10$
Numbers in Racket

• Integers in Racket are *unbounded*, i.e. they can be any size the computer can handle.

• Rational numbers are *represented exactly* as a fraction which is why \( \frac{1}{2} \) is a value not an expression.

• 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).
Expressions in Racket

Racket has many built-in functions which can be used in expressions:

- Arithmetic operators: $+, -, *, /$

- Constants: $e, \pi$

- Functions: $(\text{abs } x), (\text{max } x \ y \ \ldots), (\text{ceiling } x) \ (\text{expt } x \ y), (\text{exp } x), (\cos x), \ldots$

Look in DrRacket’s “Help Desk”. The web page that opens has many sections. The most helpful is under Teaching, then “How to Design Programs Languages”, section 1.5.
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)

\[ f(x) = x^2 \]

\[ g(x, y) = x + y \]
Defining functions (in Racket)

\[
(g \ x \ y) = x + y
\]

\[
\text{(define } \ (g \ x \ y) \ (\ + \ x \ y))\text{)}
\]
Defining functions (in Racket)

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

(define (f x) (sqr x))
(define (g x y) (+ x y))

define is a special form in Racket (i.e. it looks like a Racket function, but not all of its arguments are evaluated).

It binds (i.e. associates or pairs up) a name to an expression (which uses the parameters that follow the name).
Defining functions (in Racket)

A **function definition** in Racket consists of:

1. a *name* for the function,
2. a list of *parameters*,
3. a single *body expression* which is an expression.

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

As pointed out in the previous few slides, a function definition in Racket has the same components (in a slightly different format) as function definition in mathematics.
Applying user-defined functions in Racket

An application of a user-defined function substitutes arguments for the corresponding parameters throughout the definition’s expression.

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

The substitution for `(g 3 5)` would be `( + 3 5 )`. 
Applying user-defined functions in Racket

When faced with choices of substitutions, we use the same rules defined earlier: (1) apply functions only when all arguments are values; when you have a choice, take the leftmost one.

\[
\begin{align*}
(g \ (g \ 1 \ 3) \ (f \ 3)) & \quad \Rightarrow \quad (g \ (+ \ 1 \ 3) \ (f \ 3)) \\
& \quad \Rightarrow \quad (g \ 4 \ (f \ 3)) \\
& \quad \Rightarrow \quad (g \ 4 \ (sqr \ 3)) \\
& \quad \Rightarrow \quad (g \ 4 \ 9) \\
& \quad \Rightarrow \quad (+ \ 4 \ 9) \\
& \quad \Rightarrow \quad 13
\end{align*}
\]
Applying user-defined functions in Racket

Each parameter name has meaning only within the body of its function. I.e. *the two uses of x are independent.*

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

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

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

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

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

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

(define k 3)
(define p (sqr k))

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

In (define p (sqr k)), the expression (sqr k) is first evaluated to give 9, and then $p$ is bound to that value.
Advantages of constants

- Can give *meaningful names* to useful values (e.g. *interest-rate*, *passing-grade*, and *starting-salary*).

- *Reduces 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 (incorrectly) called variables, but their values cannot be changed (until CS 136)
Scope

The **scope** of an identifier is *where it has effect within the program.*

- The smallest enclosing scope has priority
- Can’t duplicate identifiers within the same scope

Racket Error: f: this name was defined...
Programming in DrRacket

The **definitions window**: 

- Can **save** and restore your work to/from a file 
- Can **accumulate definitions** and expressions 
- Run button **loads contents** into Interactions window 
- Provides a Stepper to let one **evaluate expressions step-by-step** 
- Features: error highlighting, subexpression highlighting, syntax checking 
- Can check the scope of a constant...
Programs in Racket: Scope

DrRacket can show what expression each identifier is bound to.

DrRacket can show what expression each identifier is bound to.
Programs in Racket

A Racket program is a sequence of definitions and expressions.

To run a program, 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.
Module 02 Summary

Some of the Components of Racket

1. **Values** are numbers or other mathematical objects. [7]
   Examples: $5, \frac{4}{9}$ (a rational number), $\pi$.

2. **Expressions** combine values with operators and functions. [7]
   Examples: $5 + 2$, $\sin(2\pi)$, $\frac{\sqrt{2}}{100\pi}$.

3. **Functions** generalize similar expressions. [8]

4. **Function definitions** consist of a name, parameters and an algebraic expression, e.g. `(define (f x y) (+ x y))`. [9]
Module 02 Summary

How to Use a Function

5. An **application** of a function supplies **arguments** for the **parameters**, which are substituted into the algebraic expression. [10]

6. Functions are evaluated by **substitution**. [11]
   (a) Functions are applied to **values**. [13]
   (b) When there is a **choice** of possible substitutions, always take the **leftmost choice**. [13]
   (c) **Result**: This substitution process ensures there is no ambiguity in the meaning of a program.
Module 02 Summary

Parenthesis, Yields and Special Forms

7. Do not include extra parentheses in an expression. [17]

8. The yields symbol, ⇒, indicates a step in the substitution process. [18]

9. A special form looks like a Racket function but not all of its arguments are evaluated. [24]

10. The special form define binds (i.e. associates) a function name with a function body. [24]
Module 02 Summary

Parameters, Scope and Constants

11. Each parameter name has meaning only within the body of its function. [28]

12. The special form define can also be used to define constants. [29]

13. Constant names can be arbitrary, so pick meaningful ones. [30]

14. An identifier’s scope is where it has effect. [31]
   (a) The smallest enclosing scope has priority. [31]
   (b) Can’t duplicate identifiers within the same scope. [31]
Module 02 Summary

A Racket Program

15. DrRacket has a **Definitions Window** to store definitions and an **Interactions Window**. [32]

16. A Racket **program** consists of a sequence of definitions and expressions. [34]

17. To run a program, the expressions are evaluated, using substitution, to produce values. [34]
Module 03: The Design Recipe

Readings:

• HtDP, section 2.5
• Survival and Style Guides

Topics:

• Programs as communication
• The design recipe
• Using the design recipe
• Tests
• Contracts
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.
Some goals for software design

We emphasize the following...

- **Correctness**: Does it give the right output? Does it meet the specification? ⇒ starting now.
- **Efficiency**: Does it minimize the use of computer resources such as processor time or memory usage? ⇒ starting 2nd year
- **Readability**: Can another programmer easily understand it?
- **Reliability**: Does it crash? Does it always work?
- **Flexibility**: Is it easy to change?
- **Extensibility**: Is it easy to add new features?
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 design recipe

• For simple programs it almost seems like you do not need to go through all this effort.

• At some point: maybe in this course, maybe in 2nd year, you’ll reach a point where you start making mistakes because the complexity of the program is too large.

• Develop good habits now.
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 of the function (*header* & *body*).

**Tests:** A *representative set of function applications* and their expected values. Examples also serve as Tests.
The design recipe

- **Purpose** is more of a *description in English* whereas the **Contract** which is more of a *mathematical description*.

- **Consume** describes the parameters.

- **Produce** is what it returns, i.e. the result of applying the function.

- The **header** is the function name, parameters, and the `define` keyword.

- The **body** is the expression that computes the result.
The design recipe

• If you want help from course staff for handling problems with your code, you need to provide them with the functions contract, purpose, and examples.

• We will be using a new function: check-expect checks to see if a function has returned the expected results.

• Generally expressions can only use functions defined before the expression. This is not the case for check-expect.
Order of execution

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

1. Write a draft of the Purpose
2. Write Examples (by hand, then code)
3. Write Definition Header & Contract
4. Finalize the purpose with parameter names
5. Write Definition Body
6. Write Tests
Using the design recipe

Purpose (first draft):

;; produce the sum of the squares of two numbers

Examples:

\[3^2 + 4^2 = 9 + 16 = 25\]

;; Example:
(check-expect (sum-of-squares 3 4) 25)
Using the design recipe (cont)

Header & Contract:

;; sum-of-squares: Num Num → Num

(define (sum-of-squares n1 n2) . . . )

Purpose (final draft):

;; (sum-of-squares n1 n2) produces the sum of the squares
;; of n1 and n2.
Using the design recipe (cont)

Write Function Body:

(define (sum-of-squares n1 n2)
  (+ (sqr n1) (sqr n2)))

Write Tests:

;; Tests:
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares −2 7) 53)
(check-expect (sum-of-squares 0 2.5) 6.25)
Using the design recipe (final result)

;; (sum-of-squares n1 n2) produces the sum of the ...
;; sum-of-squares: Num Num → Num
;; Examples:
(check-expect (sum-of-squares 3 4) 25)

(define (sum-of-squares n1 n2)
  (+ (sqr n1) (sqr n2)))

;; Tests:
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares 0 2.5) 6.25)
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 independently.
The teaching languages offer *three convenient testing methods*:

(check-expect (sum-of-squares 3 4) 25)
(check-within (sqrt 2) 1.414 .001)
(check-error (/ 1 0) "/: division by zero")

*check-within* should only be used for inexact values.

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.
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 (0, 1, 2, 3, ...)
    In this course natural numbers include 0.
  - **Any**: any Racket value

- We will see more types soon.

*Use the most specific type available.*
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.

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

In the code above, we have no idea what type is associated with the value of q without tracing through the evaluation of (mystery-fn 5).
Racket’s approach is known as **dynamic typing**, i.e. types are checked as the code is executed.

Many other mainstream languages use **static typing** in which constants, parameters and values all have specified types. Constants and parameters of one type may not hold a value of an incompatible type.

With static typing, the header of a function might look like this:

\[ \text{Int foo}(c:\text{Num}, g:\text{Nat}) \]

Here the contract is part of the language.

A program containing the function application \[ \text{foo}(65, 100.0) \] would be illegal.
Dynamic typing is a potential *source of both flexibility and confusion*, as we will see.

Contracts are important in keeping us unconfused. However, they are only human-readable comments and are not enforced by the computer.

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

Unless stated otherwise, *you may assume that all arguments provided to a function will obey the contract* (including our automated testing).
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 (pages 1–19) before completing your assignments.

The marking scheme is typically

• 5% for readability / understandability (format, meaningful identifier names, etc.)
• 5% for purpose and contract
• 10% for tests and examples, combined
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.
Module 03 Summary

The Design Recipe


2. The **header** is the function name, parameters, and the **define** keyword. [5.1]

3. The **body** is the expression that computes the result. [5.1]

4. Three new functions for testing were introduced: **check-expect**, **check-within** and **check-error**. [12]

5. Contracts may specify the following types: **Num**, **Int**, **Nat**, **Any** with more types to come later. [13]
Module 03 Summary

Requires and Types

6. Use *requires* to specify any additional constraints on the parameters not covered in the contract. [14]

7. **Dynamic typing** is when types are checked as the code is executed. [16]

8. **Static typing** is when constants, parameters and values and the values that functions produce all have specified types. [16]
Module 4: Simple Data

Readings:

• HtDP, sections 4-5

Topics:

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

A function that tests whether two numbers \( x \) and \( y \) are equal has two possible Boolean values: true and false.

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

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

Standard Racket uses \#t and \#true where we use true, and similarly for \#f, \#false, and false; these will sometimes show up in basic tests and correctness tests. You should always use true and false.
Other types of comparisons

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

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

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

;;; = : Num Num → Bool

Note that Boolean is abbreviated in contracts.
Complex relationships

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

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

These are used to test complex relationships.

Example: the proposition "\(3 \leq x < 7\)" can be computationally tested by evaluating
\[
\text{(and (\(\leq\) 3 x) (\(<\) x 7))}.
\]
Some computational differences

The mathematical AND, OR connect two propositions.

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

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

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

The function not has value true exactly when its one argument has value false.
Some computational differences

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

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

Some observations about and and or

- In some cases you do not need to evaluate every argument to know the answer
  
  \[
  (\text{and} \ \text{false} \ x) = \text{false} \quad \text{but} \quad (\text{and} \ \text{true} \ x) = x
  \]
  
  \[
  (\text{or} \ \text{true} \ x) = \text{true} \quad \text{but} \quad (\text{or} \ \text{false} \ x) = x
  \]

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

- Because not all arguments are evaluated, and and or are special forms.
Short-circuiting

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

Examples:

;; Eliminate easy cases first; might not need to do
;; the time-consuming factorization in prime?
(and (odd? x) (> x 2) (prime? x))

;; Avoid dividing by zero
(and (not (= x 0)) (<= (/ y x) c))
(or (= x 0) (> (/ y x) c))
Predicates

A predicate is a function that produces a Boolean result.

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

We can write our own:

\[
\text{(define (between? low high numb)
  (and (< low numb) (< numb high)))}
\]

\[
\text{(define (can-vote? age)
  (> = age 18))}
\]
Symbolic data

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

A symbol is defined using an apostrophe or ‘quote’: e.g. ’Earth ’Earth is a value just like 6, but it is more limited computationally.

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

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

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

Symbols can be *compared using the predicate* `symbol=?`.

```
(define home 'Earth)
(symbol=? home 'Mars) ⇒ false
```

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

- Symbols are used when you want to classify values into a few categories: e.g.
  - e.g. food: 'Chinese, 'Mexican, 'Thai, 'Italian
  - e.g. courses: 'interesting, 'ok, 'boring
  - e.g. movies: 'excellent, 'good, 'average, 'bad
- You can test to see if a result is a certain symbol using
  \( \text{(symbol=}? \, ) \)
- The contract is \textbf{symbol=}?: \text{Sym} \rightarrow \text{Bool}
Strings

Racket also supports strings, such as "blue".

What are the differences between strings and symbols?

• Strings are really *compound data*
  (i.e. a string is a *sequence* of characters).

• Symbols can’t have certain characters in them
  (such as spaces).

• More efficient to compare two symbols than two strings

• More built-in functions for strings
Strings

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

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

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

Here are a few functions which operate on strings.

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

The textbook does not use strings; it uses symbols.

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

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

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

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

Every type seen so far has an equality predicate (e.g, \(=\) for numbers, \textit{symbol} \(=?\) for symbols, \textit{string} \(=?\) for strings).

The predicate \texttt{equal?} can be used to test the equality of two values which may or may not be of the same type.

\texttt{equal?} \textit{works for almost all types of data} we have encountered so far (except inexact numbers), and most types we will encounter in the future.
When to Avoid `equal`?

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

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

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

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

Sometimes, the value an expression should take depends on some condition.

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

Example: taking the absolute value of $x$.

$$|x| = \begin{cases} 
-x & \text{when } x < 0 \\
 x & \text{when } x \geq 0 
\end{cases}$$
In Racket, we can compute $|x|$ with the expression

$$\text{(cond } [(< x 0) (- x)]
\quad [(\geq x 0) x])$$

- Conditional expressions use the special form `cond`.
- *Each argument is a question/answer pair.*
  - The question is a Boolean expression.
  - The answer is a possible value of the conditional expression.
- Square brackets used by convention, for readability.
- Square brackets and parentheses are equivalent in the teaching languages (but they must be nested properly)
- Note: `abs` is a built-in function in Racket
The *general form of a conditional expression* is ...

\[
(\text{cond \ [question1 answer1] \\
  [question2 answer2] \\
  \ldots \\
  [questionk answerk])} \ ;; \text{ where questionk could be else}
\]

- The questions are *evaluated in top-to-bottom order*.
- As soon as *one question is found that evaluates to true, no further questions are evaluated*.
- Only one answer is ever evaluated. \( \Rightarrow \) Either the one paired with the first question that evaluates to true or the one paired with else (if it is present and reached).
Example

\[ f(x) = \begin{cases} 
0 & \text{when } x = 0 \\
x \sin\left(\frac{1}{x}\right) & \text{when } x \neq 0 
\end{cases} \]

(define (f x)
  (cond [(= x 0) 0]
        [else (* x (sin (/ 1 x)))]))
Simplifying conditional functions

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

Here are the common recommendations on which course to take after CS 135, based on the mark earned.

- $0\% \leq \text{mark} < 40\%$: CS 115 is recommended
- $40\% \leq \text{mark} < 50\%$: CS 135 is recommended
- $50\% \leq \text{mark} < 60\%$: CS 116 is recommended
- $60\% \leq \text{mark}$: CS 136 is recommended
cond without simplification

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

\[
\text{(define (course-after-cs135 grade)} \\text{)}
\]
\[
\text{(cond [(< grade 40) ’cs115]}
\]
\[
\text{[(and (>= grade 40) (< grade 50)) ’cs135]}
\]
\[
\text{[(and (>= grade 50) (< grade 60)) ’cs116]}
\]
\[
\text{[(>= grade 60) ’cs136])])}
\]

This method *does not take into account* that if the computation gets to the second condition we know that the grade is greater than or equal to 40% \(\Rightarrow\) so *it can be simplified*. 
cond with simplification

We can simplify three of the tests.

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

These simplifications become second nature with practice.
Tests for conditional expressions

• Write at least *one test for each possible question/answer pair* in the expression.

• That test should be simple and direct, aimed at testing that answer.

• When the problem contains *boundary conditions* (like the cut-off between passing and failing), they *should be tested explicitly*.

• DrRacket highlights unused code.

• Properly tested code should have no highlights (i.e. no unused/untested code).
Tests for conditional expressions

For the example above:

\[
(\text{define} \ (\text{course-after-cs135} \ \text{grade})

\hspace{1cm} \ (\text{cond} \ ([(< \ \text{grade} \ 40) \ \text{’cs115}]

\hspace{1cm} \ \hspace{1cm} \ [(< \ \text{grade} \ 50) \ \text{’cs135}]

\hspace{1cm} \ \hspace{1cm} \ [(< \ \text{grade} \ 60) \ \text{’cs116}]

\hspace{1cm} \ \hspace{1cm} \ [\text{else} \ \text{’cs136}]])

\]

there are four intervals and three boundary points, so seven tests are required (for instance, 30, 40, 45 50, 55, 60, 70).
Tests for Boolean Expressions

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

For `(and (not (zero? x)) (< (\ y x) c))`, we need three test cases:

1. one test case where *x* is zero
   (first argument to **and** is **false**)

2. one test case where *x* is nonzero and \( y/x > c \),
   (first argument is **true** but second argument is **false**)

3. one test case where *x* is nonzero and \( y/x \leq c \).
   (both arguments are **true**)
Types of Tests

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 knowledge of 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 tests

The textbook writes tests in this fashion:

\[(= (\text{sum-of-squares} \ 3 \ 4) \ 25)\]

which works outside the teaching languages.

c\text{heck-expect} was added to the teaching languages after the textbook was written. You should use it for all tests.
Example: computing taxes

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

**Examples:**

Google “Canada income tax” For 2017:

- 15% on the amount in [$0 to $45,916]
- 20.5% on the amount in ($45,916 to $91,831]
- 26% on the amount in ($91,831 to $142,353]
- 29% on the amount in ($142,353 to $202,800]
- 33% on the amount over $202,800
The “piecewise linear” nature of the graph complicates the computation of tax payable.

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

This is what the paper Canadian tax form does.
<table>
<thead>
<tr>
<th>Taxable Income</th>
<th>Tax Payable</th>
<th>Income</th>
<th>Base Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Graph:**
- **Y-axis:** Tax Payable
- **X-axis:** Taxable Income
- **Points:**
  - **Breakpoint**
  - **Base Amount**
  - **Total Tax Payable**
Examples: Calculating the tax due

- 15% on the amount in [$0 to $45,916]
- 20.5% on the amount in ($45,916 to $91,831]
- ...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>0.15 \times $45,000 = $6,750</td>
</tr>
<tr>
<td>$50,000</td>
<td>0.15 \times $45,916 + 0.205 \times ($50,000 - $45,916) = $7,724.62</td>
</tr>
</tbody>
</table>

Note the notation [0 to $x$] means including $x$ and the notation ($x$ to $y$] means not including $x$. 
Example: Calculating the tax due

- 15% on the amount in [$0 to $45,916]
- 20.5% on the amount in ($45,916 to $91,831]
- 26% on the amount in ($91,831 to $142,353]
- ...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$100,000</td>
<td>$18,423.915</td>
</tr>
</tbody>
</table>

\[0.15 \times \$45,916 + 0.205 \times (\$91,831 - \$45,916) + 0.26 \times (\$100,000 - \$91,831) = \$18,423.915\]

Replace $45,916 and $91,831 with constants \(bp1\) and \(bp2\).

Replace 15%, 20.5% and 26% with constants \(rate1\), \(rate2\) and \(rate3\).
Calculating the tax due: simplification

Using these new constants the calculations become...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>rate1 × $45,000 = $6,750</td>
</tr>
<tr>
<td>$50,000</td>
<td>rate1 × bp1 + rate2 × ($50,000 - bp1) = $7,724.62</td>
</tr>
<tr>
<td>$100,000</td>
<td>rate1 × bp1 + rate2 × (bp2 - bp1) + rate3 × ($100,000 - bp2) = $18,423.915</td>
</tr>
</tbody>
</table>

Now let base1 = rate1 × bp1

and let base2 = rate1 × bp1 + rate2 × (bp2 - bp1), etc. for base3, ...
## Calculating the tax due: simplification

Using these new constants the calculations become...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>$45,000 rate1 × $45,000 = $6,750</td>
</tr>
<tr>
<td>$50,000</td>
<td>$50,000 base1 + rate2 × ($50,000 - bp1) = $7,724.62</td>
</tr>
<tr>
<td>$100,000</td>
<td>$100,000 base2 + rate3 × ($100,000 - bp2) = $18,423.915</td>
</tr>
</tbody>
</table>

With this plan in mind, we can begin to create the function that calculates the Canadian income tax for any income (i.e. for all five different rates).
### Examples:

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>$0.15 \times 45000 = 6750</td>
</tr>
<tr>
<td>$50,000</td>
<td>$0.15 \times 45916 + 0.205 \times (50000 - 45916) = 7724.62</td>
</tr>
<tr>
<td>$100,000</td>
<td>$0.15 \times 45916 + 0.205 \times (91831 - 45916) + 0.26 \times (100000 - 91831) = 18423.915</td>
</tr>
</tbody>
</table>

(check-expect (tax-payable 45000) 6750)
(check-expect (tax-payable 50000) 7724.62)
(check-expect (tax-payable 100000) 18423.915)
Definition header & contract

;; tax-payable: Num → Num
;; requires: income ≥ 0

(define (tax-payable income) . . . ...

Finalize purpose

;; (tax-payable income) computes the 2017 Canadian tax payable
;; on income.
Write function body

Some constants will be useful. Put these before the purpose and other design recipe elements.

;; Rates
(define rate1 0.15)
(define rate2 0.205)
(define rate3 0.26)
(define rate4 0.29)
(define rate5 0.33)

;; Breakpoints
(define bp1 45916)
(define bp2 91831)
(define bp3 142353)
(define bp4 202800)
Instead of putting the base amounts into the program as numbers (as the tax form does), we can compute them from the breakpoints and rates.

;; basei is the base amount for interval [bpi, bp(i+1)]
;; that is, tax payable at income bpi

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

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

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

It can be used for defining constants and the main function.
(define base1 (tax-calc 0 rate1 0 bp1))
(define base2 (tax-calc base1 rate2 bp1 bp2))
(define base3 (tax-calc base2 rate3 bp2 bp3))
(define base4 (tax-calc base3 rate4 bp3 bp4))

(define (tax-payable income)
  (cond [(< income 0) 0] ; Not strictly necessary
       [(< income bp1) (tax-calc 0 rate1 0 income)]
       [(< income bp2) (tax-calc base1 rate2 bp1 income)]
       [(< income bp3) (tax-calc base2 rate3 bp2 income)]
       [(< income bp4) (tax-calc base3 rate4 bp3 income)]
       [else (tax-calc base4 rate5 bp4 income)])))
See HtDP, section 3.1, for a good example of helper functions.

Helper functions are used for three purposes:

1. *Reduce repeated code* by generalizing similar expressions.

2. *Factor out complex calculations*.

3. *Give names to expressions*.

Style guidelines:

- Improve clarity with short definitions using well-chosen names.
- Name all functions (including helpers) meaningfully; not “helper”.
- Purpose, contract, and one example are required.
Goals of this module

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

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

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

You should be aware of other types of data (*symbols and strings*), which will be used in future lectures.
You should look for opportunities to use *helper functions* to structure your programs, and gradually learn when and where they are appropriate.
Module 04 Summary

Boolean-valued Functions

1. The Boolean values are true and false. [2]
2. Some comparison operators are: \(=, <, \leq, >, \geq\). [2-3]
3. Some Boolean connectives are: and, or, and not. [4]
4. and will produce true if all of its arguments are true. [5]
5. or will produce true if at least one of its arguments are true. [5]
6. Short-circuiting for and and or: Racket will only evaluate as many arguments as needed to determine the result. [6]
7. A predicate is a function that produces a Boolean result. [7]
Module 04 Summary

Strings and Symbols

8. Use `symbol=?` to test if one symbol equals another. [9]

9. A `strings` is a sequence of characters (i.e. text). [10]

10. There are many more operations on strings: joining, length, alphabetic order. [11]

11. Use `symbols` when you want to classify values into a few categories. [12]

12. There are many ways to check for equality: `=` for numbers, `symbol=?` for symbols, and `equal?` for any values. Use the most specific one possible. [13–14]
Module 04 Summary

Checking Conditions

13. Use `cond` to evaluate different expressions depending on the condition. [16]

14. Conditions are evaluated from the top to the bottom. [17]

15. When the first condition that evaluates to `true` is found, its corresponding expression is evaluated. [17]

16. If the condition `else` is reached, its expression will be evaluated. [17]

17. When a condition is reached, you know that all the conditions above it are false. You can use this fact to simplify the tests. [19]
Module 04 Summary

Testing

18. Test each cond question-answer pair. [22]
19. If there are boundary values, then the boundary value should be tested too. [22]
20. Tests that are not based on the details of the code (and are typically written first) are called black-box tests. [25]
21. Tests that are based on the details of the code, such as testing all the cond questions, are called white-box tests. [25]
22. Helper functions are used to reduce repeated code. [38]
Syntax & 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.

Topics:

• Modelling programming languages
• Racket’s semantic model
• Substitution rules (so far)
Modelling programming languages

• 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: ( ) , ; { } [ ] ‘ ’ “ ”

**Symbols** start with a single quote ’ followed by something obeying the rules for identifiers.
Spelling rules for Beginning Student

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

There are some built-in constants, such as 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, semantics, and ambiguity

There are three problems we need to address:

1. **Syntax**: The *form or structure* we use to say things.
   
   Not: ‘*Is This Sentence Syntactically Correct*’

2. **Semantics**: the *meaning* of what we say.
   
   Not: ‘*Trombones fly hungrily.*’

3. **Ambiguity**: valid sentences have exactly *one meaning*.
   
   Not: ‘*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, use grammars, i.e. rules used to construct valid programs (in Racket) or sentences (in English).

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:

\[
\langle \text{def} \rangle = (\text{define } (\langle \text{var} \rangle \langle \text{var} \rangle \ldots \langle \text{var} \rangle) \langle \text{exp} \rangle)
\]

There are also rules for defining constants, \textit{cond} expressions, etc.

The Help Desk presents the same idea as

\[
\text{definition} = (\text{define } (\text{id id id} \ldots) \text{ expr})
\]

In CS 135, 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.
Racket’s 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 program (in full Racket), 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.
**Application of built-in functions**

Use the rules for the arithmetic expressions 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 for a built-in function \(f\) is:

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

Note: Rules 1 and 2 were covered in Module 02 slide 13.
Note the two uses of an **ellipsis** (\(\ldots\)). What does it mean?
Ellipses

For built-in functions $f$ with one parameter, the rule is:

$$(f \ v_1) \Rightarrow v \text{ where } v \text{ is the value of } f(v_1)$$

For built-in functions $f$ with two parameters, the rule is:

$$(f \ v_1 \ v_2) \Rightarrow v \text{ where } v \text{ is the value of } f(v_1, v_2)$$

For built-in functions $f$ with three parameters, the rule is:

$$(f \ v_1 \ v_2 \ v_3) \Rightarrow v \text{ where } v \text{ is the value of } f(v_1, v_2, v_3)$$

We can’t just keep writing down rules forever, so we use *ellipses to show a pattern*:

$$(f \ v_1 \ldots \ v_n) \Rightarrow v \text{ where } v \text{ is the value of } f(v_1, \ldots, v_n).$$
Application of user-defined functions

As an example, consider \( \text{(define (term x y) (\times x (sqr y)))} \).

The \textit{function application} \( (\text{term 2 3}) \) can be evaluated by taking the body of the function definition and \textit{replacing the parameters} \( (x \text{ and } y) \) \textit{with the arguments} \( (2 \text{ and } 3) \).

The result is \( (\times 2 (sqr 3)) \).

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

Any \textit{argument} which is not a value \textit{must first be simplified to a value} using the rules for expressions.
Application of user-defined functions

The *general substitution* rule is:

4. \((f \ v_1 \ldots \ v_n) \Rightarrow \exp'\) where \((\text{define} \ (f \ x_1 \ldots \ x_n) \ \exp)\) occurs to the left, and \(\exp'\) is obtained by substituting into the expression \(\exp\), with all occurrences of the formal parameter \(x_i\) replaced by the value \(v_i\) (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.
Example:

\[
\text{define (term } x \ y) (\times x (sqr y)))
\]
\[
\text{(term } (\!-\! 3 1) (\!+\! 1 2)) \quad ;\text{ apply built-in function (to 1st arg)}
\]
\[
\Rightarrow (\text{term } 2 (\!+\! 1 2)) \quad ;\text{ apply built-in function (to 2nd arg)}
\]
\[
\Rightarrow (\text{term } 2 3) \quad ;\text{ replace parameters with arguments}
\]
\[
\Rightarrow (\times 2 (sqr 3)) \quad ;\text{ apply built-in function (to 2nd arg)}
\]
\[
\Rightarrow (\times 2 9) \quad ;\text{ apply built-in function}
\]
\[
\Rightarrow 18
\]

Note: The comments were included here to clarify each step. *On a midterm or final, do not put the comments.*
Application of user-defined constants

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

We add the substitution rule:

5. \(id \Rightarrow val\), where (define id val) occurs to the left.
Example:

\[(\text{define } x \ 3) \ (\text{define } y \ (+ \ x \ 1)) \ y\]

⇒ \hspace{2cm} \text{; substitute 3 for x}

\[(\text{define } x \ 3) \ (\text{define } y \ (+ \ 3 \ 1)) \ y\]

⇒ \hspace{2cm} \text{; apply built-in function}

\[(\text{define } x \ 3) \ (\text{define } y \ 4) \ y\]

⇒ \hspace{2cm} \text{; substitute 4 for y}

\[(\text{define } x \ 3) \ (\text{define } y \ 4) \ 4\]

Note: In order to fit this onto one slide, we have put the program onto one line and added comments, but students should avoid both of these on a midterm or final (if possible).
Module 05 Summary

Syntax, Semantics and Grammar

1. **Syntax**: The correct format (nothing unexpected or missing). [6]
2. **Semantics**: the meaning of what we say. [6]
3. **Ambiguity**: valid sentences must have exactly one meaning. [6]
4. **Grammars** are formal rules to enforce syntax and prevent ambiguity. [7]
5. A **semantic model** of a programming language specifies a method of predicting the result of running any program. [9]
6. For Racket, our model is to repeatedly simplify the program via substitution. [9]
Substitution in **cond** expressions

There are three rules: when the first expression is (1) **false**, when it is (2) **true**, and when it is (3) **else**.

6. \((\text{cond} \ [\text{false} \ \text{exp}] \ldots) \Rightarrow (\text{cond} \ldots)\).

7. \((\text{cond} \ [\text{true} \ \text{exp}] \ldots) \Rightarrow \text{exp}\).

8. \((\text{cond} \ [\text{else} \ \text{exp}]) \Rightarrow \text{exp}\).

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

Here the *ellipses* are serving a different role. They are not showing a pattern, but *showing an omission*. The rule just says “whatever else appeared after the \([\text{false} \ \text{exp}]\), you just copy it over.”
The Simplification Rules for **cond**

- $(\text{cond} \ [\text{false exp}] \ . \ . \ . ) \Rightarrow (\text{cond} \ . \ . \ . )$ means the substitution for $(\text{cond} \ [\text{false exp}] \ . \ . \ . )$ (i.e. when the first condition is false) would be $(\text{cond} \ . \ . \ . )$ (i.e. the rest of the conditions).

- $(\text{cond} \ [\text{true exp}] \ . \ . \ . ) \Rightarrow \text{exp}$ means the substitution for $(\text{cond} \ [\text{true exp}] \ . \ . \ . )$ (i.e. when the first condition is true) would be $\text{exp}$ (i.e. the expression for that condition).

- $(\text{cond} \ [\text{else exp}] \ . \ . \ . ) \Rightarrow \text{exp}$ means the substitution for $(\text{cond} \ [\text{else exp}] \ . \ . \ . )$ (i.e. the else condition) would be $\text{exp}$ (i.e. the expression for the else).
Simplification Examples for cond

• Using substitutions (cond [false exp] . . . ) ⇒ (cond . . . )
  and (cond [true exp] . . . ) ⇒ exp.

(cond [(= 1 2) (+ 0 1)] [ (< 2 3) (+ 0 2)]) ⇒

(cond [false (+ 0 1)] [ (< 2 3) (+ 0 2)] ⇒

(cond [(< 2 3) (+ 0 2)]) ⇒ ; by rule for false

(cond [true (+ 0 2)]) ⇒

(+ 0 2) ⇒ ; by rule for true

2
Simplification Examples for cond

• Using substitutions \((\text{cond} [\text{false exp}] \ldots) \Rightarrow (\text{cond} \ldots)\)
  and \((\text{cond} [\text{else exp}] \ldots) \Rightarrow \text{exp}\)

\[(\text{cond} [(= 1 2) (+ 0 1)] [\text{else} (+ 0 2)]) \Rightarrow \]

\[(\text{cond} [\text{false} (+ 0 1)] [\text{else} (+ 0 2)] \Rightarrow \]

\[(\text{cond} [\text{else} (+ 0 2)]) \Rightarrow ; \text{by rule for false} \]

\[(+ 0 2) \Rightarrow ; \text{by rule for else} \]

2
Example:

\[(\text{define } n 5) \ (\text{define } x 6) \ (\text{define } y 7)\]

\[(\text{cond } [(\text{even? } n) \ x] [(\text{odd? } n) \ y])\] ; substitute 5 for n
⇒ \[(\text{cond } [(\text{even? } 5) \ x] [(\text{odd? } n) \ y])\] ; apply built-in function
⇒ \[(\text{cond } [\text{false} \ x] [(\text{odd? } n) \ y])\] ; apply rule false cond
⇒ \[(\text{cond } [(\text{odd? } n) \ y])\] ; substitute 5 for n
⇒ \[(\text{cond } [(\text{odd? } 5) \ y])\] ; apply built-in function
⇒ \[(\text{cond } [\text{true} \ y])\] ; apply rule for true cond
⇒ y ; substitute 7 for y
⇒ 7
Example:

Similar to the previous example except here $y$ has not been defined.

```
(define n 5) (define x 6)
(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
⇒ y: this variable is not defined
```
Two types of errors

A **syntax error** occurs when *a sentence cannot be interpreted using the grammar*. Example: ) 1 ( 2 +

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

Example:

```
(cond [(> 3 4) x])
⇒ (cond [false x])
⇒ (cond )
⇒ cond: all question results were false
```
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.
Simplification rules for **and** and **or**

9. \((\text{and false . . .}) \Rightarrow \text{false.}\)

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

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

12. \((\text{or true . . .}) \Rightarrow \text{true.}\)

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

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

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

Note: In DrRacket **and** and **or** expect **two or more arguments**.
Simplification Rules for and

• For both DrRacket and CS135: \((\text{and} \ false \ . \ . \ .) \Rightarrow \text{false}\)
  but for just CS135: \((\text{and} \ true \ . \ . \ .) \Rightarrow (\text{and} \ . \ . \ .)\)
  i.e. the first argument of \text{and} gets removed.

\[(\text{and} \ (< \ 1 \ 2) \ (< \ 2 \ 3) \ (< \ 3 \ 4)) \Rightarrow \]
\[(\text{and} \ true \ (< \ 2 \ 3) \ (< \ 3 \ 4)) \Rightarrow \]
\[(\text{and} \ (< \ 2 \ 3) \ (< \ 3 \ 4)) \Rightarrow \]
\[(\text{and} \ true \ (< \ 3 \ 4)) \Rightarrow \]
\[(\text{and} \ (< \ 3 \ 4)) \Rightarrow \]
\[(\text{and} \ true) \Rightarrow \]
\[(\text{and}) \Rightarrow \text{true}\]

Use this method for CS135 clicker questions and exams.
Simplification Rules for **and**

- For both DrRacket and CS135: `(and false . . .) ⇒ false
  but for DrRacket: `(and true . . .) ⇒ `(and true . . .)

  i.e. the first argument of `and does not` get removed. E.g.

  `(and (< 1 2) (< 2 3) (< 3 4)) ⇒ `(and true (< 2 3) (< 3 4)) ⇒ `(and true true (< 3 4)) ⇒ `(and true true true) ⇒ true

**Do not** use this method for CS135 clicker questions and exams.
Simplification Rules for **or**

- For both DrRacket and CS135: \((\text{or} \ \text{true} \ . \ . \ .) \Rightarrow \text{true}\)

  but for CS135: \((\text{or} \ \text{false} \ . \ . \ .) \Rightarrow (\text{or} \ . \ . \ .)\)

  i.e. the first argument of **or** gets removed. E.g.

  \((\text{or} \ (= \ 1 \ 2) \ (= \ 2 \ 3) \ (< \ 3 \ 4)) \Rightarrow\)

  \((\text{or} \ \text{false} \ (= \ 2 \ 3) \ (< \ 3 \ 4)) \Rightarrow\)

  \((\text{or} \ (= \ 2 \ 3) \ (< \ 3 \ 4)) \Rightarrow\)

  \((\text{or} \ \text{false} \ (< \ 3 \ 4)) \Rightarrow\)

  \((\text{or} \ (< \ 3 \ 4)) \Rightarrow\)

  \((\text{or} \ \text{true}) \Rightarrow\)

  \text{true}

Use this method for CS135 clicker questions and exams.
Simplification Rules for or

- For both DrRacket and CS135: \((\text{or false} \ldots) \Rightarrow \text{false}\)
  
  but for DrRacket: \((\text{or false} \ldots) \Rightarrow (\text{or false} \ldots)\)

  i.e. the first argument of or does not get removed. E.g.

  \[
  (\text{or } (= 1 2) (= 2 3) (< 3 4)) \Rightarrow
  \]

  \[
  (\text{or false } (= 2 3) (< 3 4)) \Rightarrow
  \]

  \[
  (\text{or false false } (< 3 4)) \Rightarrow
  \]

  \[
  (\text{or false false true}) \Rightarrow
  \]

  \[
  \text{true}
  \]

Do not use this method for CS135 clicker questions and exams.
**Substitution rules** (so far)

1. Apply functions only when all arguments are values. [Mod 02:13]

2. When given a choice, evaluate the leftmost expression first. [Mod 02:13]

3. \((f \, v_1...v_n) \Rightarrow v\) when \(f\) is built-in function... [10-11]

4. \((f \, v_1...v_n) \Rightarrow exp'\) when \((\text{define} \, (f \, x_1...x_n) \, \text{exp})\) occurs to the left... [12–14]

5. \(id \Rightarrow \text{val}\) when \((\text{define} \, id \, \text{val})\) occurs to the left. [15–16]
6. \((\text{cond [false exp] . . .}) \Rightarrow (\text{cond . . .})\) [17–19].

7. \((\text{cond [true exp] . . .}) \Rightarrow \text{exp}\) [17–19].

8. \((\text{cond [else exp]}) \Rightarrow \text{exp}\) [17–19].

9. \((\text{and false . . .}) \Rightarrow \text{false}\) [21–22].

10. \((\text{and true . . .}) \Rightarrow (\text{and . . .})\) [21–22].

11. \((\text{and}) \Rightarrow \text{true}\) [21–22].

12. \((\text{or true . . .}) \Rightarrow \text{true}\) [21–22].

13. \((\text{or false . . .}) \Rightarrow (\text{or . . .})\) [21–22].

14. \((\text{or}) \Rightarrow \text{false}\). [21–22].
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.
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.
Module 05 Summary

Syntax, Semantics and Grammar

1. **Syntax:** The correct format (nothing unexpected or missing). [6]
2. **Semantics:** the meaning of what we say. [6]
3. **Ambiguity:** valid sentences must have exactly one meaning. [6]
4. **Grammars** are formal rules to enforce syntax and prevent ambiguity. [7]
5. A **semantic model** of a programming language specifies a method of predicting the result of running any program. [9]
6. For Racket, our model is to repeatedly simplify the program via substitution. [9]
Module 05 Summary

Errors

7. **Ellipsis** are used to show *several arguments* [11] or to show *omissions*, e.g. the rest of the conditions. [17]

8. A **syntax error** occurs when a sentence cannot be interpreted using the grammar (i.e. it is not in a correct format). [20]

9. A **run-time error** occurs when an expression cannot be reduced to a value by application of our (still incomplete) evaluation rules. [20]
Module 05 Summary

Substitution Rules

10. Our substitution rules for and and or are different from DrRacket’s rules. [22]

11. Know the **Substitution Rules** [23-24] and how to apply them.

12. Doing a step-by-step reduction using to these rules is called **tracing a program**. [25]
Lists

Readings: HtDP, sections 9 and 10.

- Avoid 10.3 (uses draw.ss).
- The textbook introduces “structures” before lists. The discussion of lists makes a few references to structures that can be ignored.
Topics:

- Introducing lists [3–13]
- Contracts involving lists [14–18]
- Processing lists: Data definitions & templates [19–40]
- Patterns of recursion [41–43]
- Producing lists from lists [44–48]
- Design recipe refinements [49–51]
- Strings and lists of characters [52-55]
- Wrapper functions [56-58]
Introducing lists

Numbers and Boolean values represent a single item.

But there are many circumstances in which we need to track many items: the names of all the students in a course, the weight of each bag loaded on an airplane, etc.

Furthermore, the amount of data *may change over time* (grow or shrink) without a predetermined maximum size (i.e. its size is unbounded).

The order of values may also be important.

Many programming languages meet this need with lists.
A list is a **recursive structure** – it is *defined in terms of a smaller structure* (i.e. a smaller list). Consider a list of concerts:

- 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**.

*Lists are created with* `(cons aValue aList)`, which adds `aValue` to the beginning of `aList`. The constant **empty** is the empty list.
Example lists with 0, 1 or 2 items

A sad state of affairs – no upcoming concerts to attend:

```
(define concerts0 empty)
```

A list with one concert to attend:

```
(define concerts1 (cons "Waterboys" empty))
```

A new list just like concerts1 but with a new concert at the beginning:

```
(define concerts2 (cons "DaCapo" concerts1))
```
Example lists with 2 or 3 items

Another way to write concerts2:

\[
\text{(define concerts2alt (cons "DaCapo"}
\quad (\text{cons "Waterboys"}
\quad \text{empty}))}
\]

A list with one U2 and two DaCapo concerts:

\[
\text{(define concerts3 (cons "U2"}
\quad (\text{cons "DaCapo"}
\quad (\text{cons "DaCapo"}
\quad \text{empty}))}
\]
Basic list constructs

- **empty**: A value representing an empty list.
- **(cons aValue aList)**: Consumes aValue and aList producing a new, longer list.
- **(first lst)**: Consumes a nonempty list; produces the first value.
- **(rest lst)**: Consumes a nonempty list; produces the same list without the first value.
- **(empty? aValue)**: Consumes aValue; produces true if it is empty and false otherwise.
- **(cons? aValue)**: Consumes aValue; produces true if it is a cons value and false otherwise.
- **(list? v)**: Equivalent to (or (cons? v) (empty? v)).
Extracting values from a list

\[(\text{define clst} \ (\text{cons} \ "\text{Waterboys}\)"
\ (\text{cons} \ "\text{DaCapo}\)" \ (\text{cons} \ "\text{Waterboys}\)" \ \text{empty})))\]

First concert:
\[(\text{first clst}) \Rightarrow "\text{Waterboys}"\]

Concerts after the first:
\[(\text{rest clst}) \Rightarrow (\text{cons} \ "\text{DaCapo}\)" \ (\text{cons} \ "\text{Waterboys}\)" \ \text{empty})]\]

Second concert:
\[(\text{first (rest clst)}) \Rightarrow "\text{DaCapo}"\]
**Nested boxes visualization**

`cons` can be thought of as producing value with two parts.

It can be visualized two ways. The first:

\[
(\text{cons} \ "\text{Waterboys}\" \ \text{empty})
\]

```
Waterboys   empty
```

(\text{cons} \ "\text{DaCapo}\" \ (\text{cons} \ "\text{Waterboys}\" \ \text{empty}))

```
first  rest
first  rest
Waterboys
Waterboys   empty
```

CS 135 Fall 2019  06: Lists
(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)))
Simple functions on lists: next-concert

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

;; (next-concert loc) produces the next concert to attend or false if loc is empty

(check-expect (next-concert (cons "a" (cons "b" empty))) "a")
(check-expect (next-concert empty) false)

(define (next-concert loc)
  (cond [(empty? loc) false]
        [else (first loc)]))
Simple functions on lists: \texttt{same-consec}

\begin{quote}
\texttt{;; (same-consec? loc) determines if next two concerts are the same}
\end{quote}

\begin{quote}
\texttt{(check-expect (same-consec? (cons "a" (cons "b" empty))) false)}
\end{quote}

\begin{quote}
\texttt{(check-expect (same-consec? (cons "a" (cons "a" empty))) true)}
\end{quote}

\begin{quote}
\texttt{(check-expect (same-consec? (cons "a" empty)) false)}
\end{quote}

\begin{verbatim}
(define (same-consec? loc)
  (and (not (empty? loc))
       (not (empty? (rest loc)))
       (string=? (first loc) (first (rest loc)))))
\end{verbatim}
Contracts involving lists

What is the contract for (next-concert loc)?

We could use “List” for loc.

However, we almost always need to answer the question list of what:
A list of numbers? A list of strings? A list of any type at all?
(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 strings and has type (listof Str).

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

*Replace X with the most specific type available.*

(listof X) always includes the empty list, empty.
(anyof ... ) notation in contracts

What about the value produced by next-concert? It might be a string or it might be false.

Use (anyof X Y ...) to mean any of the listed types or values.

Examples:

- (anyof Num Str)
- (anyof Str Num Bool)
- (anyof 1 2 3)
- (listof (anyof Str 'SomeSymbol))
Syntax for lists

*Lists are values* (i.e. they cannot be simplified).

The following are valid expressions:

- `empty`
- `(cons aValue aList)`
- `(first (cons aValue aList))`
- `(rest (cons aValue aList))`
- `(empty? e)`
- `(cons? e)`

Here `e` is any Racket expression, `aValue` is any Racket value (including lists) and `aList` is a list (which includes `empty`).
Semantics for lists

The substitution rules are:

- \((\text{first} \ (\text{cons} \ a\text{Value} \ a\text{List})) \Rightarrow a\text{Value}\).
- \((\text{rest} \ (\text{cons} \ a\text{Value} \ a\text{List})) \Rightarrow a\text{List}\).
- \((\text{empty}\text{-}? \ \text{empty}) \Rightarrow \text{true}\).
- \((\text{empty}\text{-}? \ a\text{Value}) \Rightarrow \text{false}, \text{ where } a\text{Value} \text{ is any Racket value other than empty.}\)
- \((\text{cons}\text{-}? \ (\text{cons} \ a\text{Value} \ a\text{List})) \Rightarrow \text{true}\).
- \((\text{cons}\text{-}? \ a\text{Value}) \Rightarrow \text{false}, \text{ where } a\text{Value} \text{ is any Racket value not created using cons.}\)
Processing lists: data defs & templates

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?

*The structure of a function often mirrors the structure of the data it consumes.* As we encounter more complex data types, we will find it useful to be precise about their structures.

We will do this by developing **Data Definitions**.

We can even go so far as developing function **Templates** based on the data definitions of the values it consumes.
List data definition

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

;; A (listof Str) is one of:

;; ★ empty

;; ★ (cons Str (listof Str))

This is a **recursive data definition**, i.e. it has a *base case* and a **recursive (self-referential) case**.

We can use this data definition to show rigourously that (cons "a" (cons "b" empty)) is a (listof Str).
We can generalize lists of strings to other types by using an $X$:

;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))
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.
Template for processing a `(listof X)`

We start with the data definition for a `(listof X)`: 

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

A function consuming a `(listof X)` will need to distinguish between these two cases.
Template for processing a \((\text{listof } X)\)

Data Definition

;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))

Template

;; listof-X-template: (listof X) \rightarrow \text{Any}
(define (listof-X-template lox)
  (cond [(empty? lox) . . . ]
        [(cons? lox) . . . ]))

The . . . represents a place to \textit{fill in code} for what expression to return in the case of an empty or a non-empty list.

In the last case (i.e. \((\text{cons? } \text{lox})\)) we \textit{know} from the data definition that there is a first \(X\) and the rest of the list of \(X\)’s, so ...
Template for processing a (listof X) refined

;; listof-X-template: (listof X) → Any
(define (listof-X-template lox)
  (cond [(empty? lox) . . . ]
        [(cons? lox) (. . . (first lox) . . . (rest lox) . . . )]))

Now we go a step further.

Because (rest lox) is of type (listof X), we apply the same computation to it – that is, we apply listof-X-template.
Template for processing a `(listof X)` refined

;; listof-X-template: (listof X) → Any
(define (listof-X-template lox)
  (cond [(empty? lox) ... ]
        [(cons? lox) (... (first lox) ...
                        (listof-X-template (rest lox)) ... )]))

This is the template for a function consuming a `(listof X)`. Its form parallels the data definition.

We can now fill in the dots for a specific example – counting the number of concerts in a list.
Example: how many concerts?

We begin with writing the purpose, examples, contract, and then copying the template and renaming the function and parameters.

;;; (count-concerts loc) counts the number of concerts in loc
;;; count-concerts: (listof Str) → Nat
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)
(define (count-concerts loc)
  (cond [(empty? loc) . . . ]
        [else (. . . (first loc) . . .
                  . . . (count-concerts (rest loc)) . . . )]))
Thinking about list functions

Here are three crucial questions to help think about functions consuming a list:

1. What does the function produce in the *base case*?

2. What does the function produce for the *first element* in a non-empty list?

3. How does the function *combine* the value produced from the *first element* with the value obtained by applying the function to the *rest of the list*?
Example: how many concerts?

;; (count-concerts los) counts the number of concerts in los
;; count-concerts: (listof Str) → 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)))]))

The only parts not in the template are: 0 and + 1.

This is a recursive function (it uses recursion).
Recursive Functions

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

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)))
⇒ (+ 1 (count-concerts (cons "b" empty)))
⇒ (+ 1 (+ 1 (count-concerts empty)))
⇒ (+ 1 (+ 1 0))
⇒ 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.
Use of a condensed trace

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 Str) → 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) ...)
Generalizing: count-waterboys

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 string to be counted.

;; count-string: Str (listof Str) → Nat
(define (count-string s los) . . . )

The recursive function application will be (count-string s (rest los)).
More about list templates

Here are three crucial questions to help think about functions consuming a list and filling in their templates:

• What does the function produce in the *base case*? Fill in that part of the template.

• What does the function do to the *first element in a non-empty list*? Fill in that part of the template.

• How does the function *combine* the value produced from the *first element* with the value obtained by applying the function to the *rest of the list*?
Refining the \textbf{(listof X)} template

Sometimes, each \textit{X} in a (listof X) may require further processing. Indicate this with a template for \textit{X} as a helper function.

\[
;;\;\text{listof-X-template: (listof X) } \rightarrow \text{ Any}
\]

\[\begin{aligned}
(\text{define } & (\text{listof-X-template } \text{lox}) \\
& (\text{cond } [(\text{empty? } \text{lox}) \ldots ] \\
& \quad [\text{else } \ldots (\text{X-template (first lox)}) \ldots \\
& \quad \quad \ldots (\text{listof-X-template (rest lox)}) \ldots )])\]
\end{aligned}\]

\textit{We assume this generic data definition and template from now on.}
Templates as generalizations

*Key Point:* 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.
Patterns of recursion

The list template has the property that \textit{the form of the code matches the form of the data definition}.

We will call this \textbf{simple recursion}.

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

Until we do, the functions we write (and ask you to write) will use simple recursion (and hence will fit the form described by such templates).

Use the templates.
Simple recursion

A clearer definition: In simple recursion, every argument in a recursive function application is either:

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

(define (func lst) ... (func (rest lst)) ...) ;; Simple
(define (func lst x) ... (func (rest lst) x) ...) ;; Simple
(define (func lst x) ... (func (process lst) x) ...) ;; NOT Simple
(define (func lst x)
  ... (func (rest lst) (math-function x)) ...) ;; NOT Simple
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) → (listof Num)
(\text{check-expect (negate-list empty) empty})
(\text{check-expect (negate-list (cons 2 (cons −12 empty)))})
\hspace{1cm} (\text{cons −2 (cons 12 empty)})

Since **negate-list** consumes a \textbf{(listof Num)}, we use the general list template to write it.
Example: negate-list with template

;; (negate-list lon) produces a list with every number in lon negated
;; negate-list: (listof Num) → (listof Num)
;; Examples:

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

(define (negate-list lon)
  (cond [(empty? lon) . . .]
        [else (. . . (first lon) . . . (negate-list (rest lon)) . . . )])))
Example: 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

\[(\text{negate-list} \ (\text{cons} \ 2 \ (\text{cons} \ -12 \ \text{empty}))))\]
\[\Rightarrow \ (\text{cons} \ (-2) \ (\text{negate-list} \ (\text{cons} \ -12 \ \text{empty}))))\]
\[\Rightarrow \ (\text{cons} \ -2 \ (\text{negate-list} \ (\text{cons} \ -12 \ \text{empty}))))\]
\[\Rightarrow \ (\text{cons} \ -2 \ (\text{cons} \ (-12) \ (\text{negate-list} \ \text{empty}))))\]
\[\Rightarrow \ (\text{cons} \ -2 \ (\text{cons} \ 12 \ (\text{negate-list} \ \text{empty}))))\]
\[\Rightarrow \ (\text{cons} \ -2 \ (\text{cons} \ 12 \ \text{empty})))\]
Nonempty lists

**Key Point:** Sometimes a given computation makes sense only on a nonempty list.

For example, finding the maximum of a list of numbers.

**Exercises:**

Create a self-referential data definition for \((\text{ne-listof } X)\), a nonempty list of \(X\).

Develop a template for a function that consumes an \((\text{ne-listof } X)\).

Finally, write a function to find the maximum of a nonempty list of numbers.
Design recipe refinements

When we introduce new types, like (ne-listof X), we need to include it in the design recipe.

*Key Point:* For each new type, place the following someplace between the top of the program and the first place the new type is used:

- The data definition
- The template derived from that data definition

This information is only needed once.
Data definitions

Example:

;; A (listof X) is one of:
;; ★ empty
;; ★ (cons X (listof X))

In a self-referential data definition, at least one clause (possibly more) must not refer back to the definition itself; these are base cases.

Assignments do not need to include data definitions or templates for (listof X). You do for (ne-listof X) and other types you may define.
Templates

The template follows directly from the data definition.

Key Point: The overall shape of a self-referential 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.

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

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

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

*Key Point:* Racket’s notation for the character ‘a’ is \#\textbackslash\textbackslash{}a.

The result of evaluating \((\text{string} \rightarrow \text{list} \ "\text{test}\" )\) is the list \((\text{cons} \ \#\textbackslash\textbackslash{}t \ (\text{cons} \ \#\textbackslash\textbackslash{}e \ (\text{cons} \ \#\textbackslash\textbackslash{}s \ (\text{cons} \ \#\textbackslash\textbackslash{}t \ \text{empty})))))\).

This is unfortunately ugly, but the \# notation is part of a more general way of specifying values in Racket.
Example: Counting characters in a string

Write a function to count the number of occurrences of a specified character in a string. Start by counting the occurrences in a list of characters.

;; (count-char/list ch loc) counts the number of occurrences of ch in loc.
;; count-char/list: Char (listof Char) → Nat
(check-expect (count-char/list #\e (string→list " ")) 0)
(check-expect (count-char/list #\e (string→list "beekeeper")) 5)
(check-expect (count-char/list #\o (cons #\f (cons #\o (cons #\o (cons #\d empty)))))) 2)
;; (count-char/list ch loc) counts the number of occurrences of ch in loc.
;; count-char/list: Char (listof Char) → Nat

(check-expect (count-char/list #\e (string→list "")) 0)
(check-expect (count-char/list #\e (string→list "beekeeper")) 5)

(define (count-char/list ch loc)
  (cond [(empty? loc) 0]
        [else (+ (cond [(char=? ch (first loc)) 1]
                        [else 0])
                (count-char/list ch (rest loc))))))
Wrapper functions

Our functions should be easy to use. The problem statement was to count characters in a string, not in a list of characters.

We shouldn’t expect the user of our function to know that to use count-char/list they need to convert their string to a list of characters.

In such cases it’s good practise to include a wrapper function – a simple function that “wraps” the main function and takes care of housekeeping details like converting the string to a list.
;; (count-char ch s) counts the number of occurrences
;; of ch in s.
;; count-char: Char Str → Nat

(check-expect (count-char #\e " " ) 0)
(check-expect (count-char #\e "beekeeper") 5)

(define (count-char ch s)
  (count-char/list ch (string→list s)))
Characteristics of wrapper functions

Wrapper functions:

• are short and simple

• always call another function that does much more

• *sets up the appropriate conditions for calling the other function*, usually by transforming one or more of its parameters or providing a starting value for one of its arguments
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 \((\text{listof } X)\) 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).
Module 06 Summary

Lists

1. Two functions to create lists: empty and cons (i.e. construct). [4]
2. Two functions to check lists: empty? and cons? (e.g. is it non-empty). [7]
3. Two functions to break-up a list into two parts: first and rest. [7]
4. Use (listof X) in contracts, where X is one of Num, Bool, Sym, Any etc. [15]
5. Use anyof to list the mixed types explicitly in a contract. [16]
6. (cons a b) is a value, i.e. it won’t be simplified. [17]
7. For (cons a b), a must be a value and b must be a list. [17]
Module 06 Summary

Recursion, Templates and Traces

8. A **recursive definition** consists of a **base case** and a recursive (i.e. **self-referential**) case. [20]

9. A **template** is a general framework within which we fill in specifics. [22]

10. A function is **recursive** when the body of the function involves an application of the same function. [30]

11. We will typically use a **condensed trace** where we just show the results of the application of each recursive function call (rather than substitute the body of the function). [33-35]
Module 06 Summary

Design Recipe and Templates

12. For recursive list functions,
   (a) start with cond answers for the base cases,
   (b) then consider the first element
   (c) and finally how to combine the answers from first and rest of
       the list. [38]


14. Create a self-referential data definition and template for each
    new self-referential type. [49]

15. Then follow the Design Recipe for each function. [51]
Module 06 Summary

Simple Recursion

16. For **simple recursion** the form of the code matches the form of the data definition. [41]

17. For **simple recursion**, every argument is either
   (a) unchanged, [42]
   (b) one step closer to a base case. [42]

18. Two list functions are **member?** and **length**. [43]

19. The base case for the data definition of a non-empty list of Nums would be **(cons Num empty)** rather than **empty**. [48]
Module 06 Summary

Strings and Wrapper Functions

20. A string can be thought of as a list of characters. [45]

21. Racket provides two functions `string->list` and `list->string` to convert back and forth. [52]

22. The Racket notation for the character ‘a’ is `#\a`. [50]

23. A **wrapper function** is a small function that modifies the arguments passed to a **primary function**. [58]
Natural numbers – recursively

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

Topics:

• Review: data def and templates
• Natural numbers: data def and templates
• Subintervals
• Counting up
Review: from definition to template

Key Idea: modify our approach for dealing with lists so that we can deal with natural numbers.

First, we’ll review how we derived the list template.

;; A (listof X) is one of:

;; ★ empty

;; ★ (cons X (listof X))

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

*Key Idea:* We must consider (1) a base case (2) a single natural number and (3) the rest of the natural numbers.
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 \((\text{zero? } n)\) tells us which case applies.

If \((\text{zero? } n)\) is false, then \( n \) has the value \((\text{add1 } k)\) for some \( k \).

To compute \( k \), we subtract 1 from \( n \), using the built-in \text{sub1} function.

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

\[
\text{(define (nat-template } n) \\
\quad (\text{cond } [(\text{zero? } n) \ldots ] \\
\quad \quad [\text{else } \ldots n \ldots \\
\quad \quad \quad \ldots (\text{nat-template } (\text{sub1 } n)) \ldots ]))
\]
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)))

With these examples, we proceed by filling in the template.
;; (countdown n) produces a decreasing list of Nats from n to 0
;; countdown: Nat → (listof Nat)
(check-expect (countdown 0) (cons 0 empty))
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

(define (countdown n)
  (cond [(zero? n) . . . ]
        [else (. . . n . . . (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 decreasing list of Nats from n to 0
;; countdown: Nat → (listof Nat)

;; Example:
(check-expect (countdown 0) (cons 0 empty))
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

(define (countdown n)
  (cond [(zero? n) (cons 0 empty)]
        [else (cons n (countdown (sub1 n)))]))
A condensed trace

(countdown 2)
⇒ (cons 2 (countdown 1))
⇒ (cons 2 (cons 1 (countdown 0)))
⇒ (cons 2 (cons 1 (cons 0 empty)))
⇒ (cons 2 (cons 1 (cons 0 empty)))
Subintervals of Integers

*Key Point:* with a slight modification we extend our approach to talk about ranges of integers.

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

We can add subscripts to define subsets (a.k.a. ranges) of the integers.

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

Other examples: $\mathbb{Z}_{>4}, \mathbb{Z}_{<-8}, \mathbb{Z}_{\leq 1}$.
Non-zero base case: E.g. \( \mathbb{Z}_{\geq 7} \)

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:

```scheme
;; An integer in \( \mathbb{Z}_{\geq 7} \) is one of:

;;; \( \star \)

;;; \( \star \) 7

;;; \( \star \) (add1 \( \mathbb{Z}_{\geq 7} \))
```

We use this data definition as a guide when writing functions, but in practice we use a requires section in the contact to capture the new stopping point.
;; (countdown-to-7 n) produces a decreasing list from n to 7
;; countdown-to-7: Nat → (listof Nat)
;; requires: n ≥ 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)))]))

Note: in the Data Definition we add1 up from our base case but in the template we sub1 down to our base.
Generalizing **countdown** and **countdown-to-7**

*Key Point:* 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” (be passed unchanged) in the recursion.
(define (countdown-to n b)
  (cond [(= n b) (cons b empty)]
        [else (cons n (countdown-to (sub1 n) b))])))
Another condensed trace

\( (\text{countdown-to} \ 4 \ 2) \)

⇒ \( (\text{cons} \ 4 \ (\text{countdown-to} \ 3 \ 2)) \)

⇒ \( (\text{cons} \ 4 \ (\text{cons} \ 3 \ (\text{countdown-to} \ 2 \ 2))) \)

⇒ \( (\text{cons} \ 4 \ (\text{cons} \ 3 \ (\text{cons} \ 2 \ \text{empty}))) \)
countdown-to with negative numbers

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

(countdown-to 1 −2)
⇒ (cons 1 (cons 0 (cons −1 (cons −2 empty)))))
Counting up: basic idea

*Key Point:* 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.
Counting up template

Notice the *additional requires section* capturing the restriction on n.

;; nonpos-template: Int → Any
;; requires: n ≤ 0

(define (nonpos-template n)
  (cond [(zero? n) . . .]
        [else (. . . n . . .
            [else ( . . n . . .
                . . . (nonpos-template (add1 n)) . . . )]
          )]
    ))

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

(cons −2 (cons −1 (cons 0 empty))).
Counting up example

;; (countup n) produces an increasing list from n to 0
;; countup: Int → (listof Int)
;; requires: n ≤ 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))))]))
Counting up to $b$

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

;;; (countup-to n b) produces an increasing list from n to b
;;; countup-to: Int Int → (listof Int)
;;; requires: n ≤ 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))]))
Comparisons with imperative programming

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

```plaintext
define loop
  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.
Data definitions vs. templates

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.

*Recall* that in the Data Definition, recursion moves *away from the base case*. In the template, recursion moves *towards the base case*.

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.
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.
Module 07 Summary

Natural Numbers and Integers

1. There is a recursive definition for the set of natural numbers, Nat (which is similar to the one for lists). [5]
2. The functions add1, sub1 and zero? are used with Nat’s. [5–6]
3. This is also a recursive definition for ranges of integers with a finite end point. [11]
4. Use $\mathbb{Z}_\geq i$ to specify integers greater than or equal to $i$. [11]
5. We can count down to an integer [11–17] or count up. [18–21]
6. In the template, recursion moves towards the base. In the data definition, recursion moves away from the base. [23]
More lists

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

Topics:

- Sorting a list [2–9]
- List abbreviations [10–14]
- Lists containing lists [15–29]
- Dictionaries and association lists [30–38]
- Lists of lists as 2D data [39–41]
- Processing two lists simultaneously [42–60]
- Consuming a list and a number [61–67]
- List equality [68-71]
Sorting a list

When writing a function to consume a list, we may find that we need to create an helper function to do some of the work.

*Key Point:* The helper function may or may not be recursive itself.

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.
The list template

;; (sort lon) sorts the elements of lon in nondecreasing order
;; sort: (listof Num) → (listof Num)
(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) . . . )

(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.
Filling in the list template

;; (sort lon) sorts the elements of lon in nondecreasing order
;; sort: (listof Num) → (listof Num)
(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) . . . )

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

Approach: insert (which we will implement) will be a recursive helper function that consumes a number and a sorted list. It inserts the number into 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 3 into empty list
⇒ (insert 2 (insert 4 (cons 3 empty))) ; insert 4
⇒ (insert 2 (cons 3 (cons 4 empty))) ; insert 2
⇒ (cons 2 (cons 3 (cons 4 empty))) ; sorted list
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) → (listof Num)
;; requires: slon is sorted in nondecreasing order
(define (insert n slon)
  (cond [(empty? slon) . . . ]
      [else (. . . (first slon) . . .
               (insert n (rest slon)) . . . )]))
Filling out the template

Recall that filling out the template is a matter of providing answers to certain questions.

If \texttt{slon} is empty, the result is the list containing just \texttt{n}.

If \texttt{slon} is not empty, another conditional expression is needed. \texttt{n} is the first number in the result if it is less than or equal to the first number in \texttt{slon}.

Otherwise, the first number in the result is the first number in \texttt{slon}, and the rest of the result is what we get when we insert \texttt{n} into \texttt{(rest slon)}.
The implementation of \texttt{insert}

\begin{verbatim}
(define (insert n slon)
  (cond [(empty? slon) (cons n empty)]
    [(<= n (first slon)) (cons n slon)]
    [else (cons (first slon) (insert n (rest slon)))]))
\end{verbatim}
A condensed trace of insert

\[
\begin{align*}
\text{(insert } 4 \ (\text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 5 \ \text{empty})))) & \Rightarrow \ (\text{cons } 1 \ (\text{insert } 4 \ (\text{cons } 2 \ (\text{cons } 5 \ \text{empty})))) \\
& \Rightarrow \ (\text{cons } 1 \ (\text{cons } 2 \ (\text{insert } 4 \ (\text{cons } 5 \ \text{empty})))) \\
& \Rightarrow \ (\text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 4 \ (\text{cons } 5 \ \text{empty})))) \\
\end{align*}
\]

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

*Key Idea:* Now that we understand lists, we can abbreviate them.

In DrRacket, “Beginning Student With List Abbreviations” provides new syntax for list abbreviations, and a number of additional convenience functions.

Remember to follow the instructions in Module 01 when changing language levels.
List abbreviation 1: list

The expression

(cons exp1 (cons exp2 (\ldots (cons expn empty)\ldots )\ldots ))

can be abbreviated as

(list exp1 exp2 \ldots expn)

The result of the trace we did on the last slide can be expressed as

(list 1 2 4 5).
Accessing elements of a list

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

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

Key Point: Use these sparingly to improve readability.

The templates we have developed remain very useful.
**cons vs. list**

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 longer 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).
List abbreviation 2: Quoting lists

If lists built using `list` consist of *just symbols, strings, and numbers*, the list can be abbreviated using the quote notation.

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

`(list “a” “b” “c”)` can be written as ‘(“a” “b” “c”)’ because quoted strings evaluate to strings.

`(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

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

Here are two different two-element lists.

1 2
3 4
(cons 1 (cons 2 empty))
(cons 3 (cons 4 empty))
Example: a list containing another list

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

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

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

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

\[(\text{cons} \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ \text{empty}))) \ (\text{cons} \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ \text{empty})) \ \text{empty})]\]
Expressing Nested Lists

We have several ways of expressing this list in Racket:

\[(\text{cons} \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ \text{empty})) \ \\
\quad \ (\text{cons} \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ \text{empty})) \ \\
\quad \quad \ \text{empty}))\]

\[(\text{list} \ (\text{list} \ 1 \ 2) \ (\text{list} \ 3 \ 4))\]

\['(\text{list} \ (1 \ 2) \ (3 \ 4))\]

Clearly, the abbreviations are more expressive.
Example: taxes

A company needs to process their payroll – a list of employee names and their salaries. It produces a list of each employee name and the tax owed. The tax owed is computed with tax-payable from Module 04.

Payroll:

(list (list "Asha" 50000)
 (list "Joseph" 100000)
 (list "Sami" 10000))

TaxOwed:

(list (list "Asha" 7724.62)
 (list "Joseph" 18423.915)
 (list "Sami" 1500))
Data definitions for taxes

;; A Payroll is one of:
;; * empty
;; * (cons (list Str Num) Payroll)

;; A TaxOwed is one of:
;; * empty
;; * (cons (list Str Num) TaxOwed)

Note: Both Payroll and TaxOwed are (listof X) where X is a two-element list.
Template: preliminary version

;; (payroll-template pr)

;; payroll-template: Payroll → Any

(define (payroll-template pr)
  (cond [(empty? pr) . . . ]
    [(cons? pr) . . . (first pr) . . .
      . . . (payroll-template (rest pr)) . . . ])

A payroll is just a list, so it looks exactly like the (listof X) template – so far...
Template: refinements

*Key Point:* Some information from our data definition is not yet captured in the template: The list’s first item is known to be of the form \((\text{list Str Num})\).

It’s useful to reflect that fact in the template:

- It reminds us of all the data available to us when solving the problem.
- Our solutions (derived from the template) will often access the parts of the sublist.
Template: final version

;; (payroll-template pr)

;; payroll-template: Payroll → Any

(define (payroll-template pr)
  (cond [(empty? pr) . . . ]
        [(cons? pr) (. . . (first (first pr)) . . .
                      . . . (first (rest (first pr))) . . .
                      . . . (payroll-template (rest pr)) . . . )]]

Some short helper functions (namely name and amount) will make our code more readable.
;; (name lst) produces the first item from lst – the name.
(define (name lst) (first lst))

;; (amount lst) produces the second item from lst – the amount.
(define (amount lst) (first (rest lst)))

;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) . . . ]
        [(cons? pr) (. . . (name (first pr)) . . .
                      . . . (amount (first pr)) . . .
                      . . . (payroll-template (rest pr)) . . . )])))

*Note:* Non-recursive helper functions only need a purpose.
Using the design recipe: fill in template

;; (compute-taxes payroll) calculates the tax owed for each
;; employee/salary pair in the payroll.
;; compute-taxes: Payroll → TaxOwed

(check-expect (compute-taxes test-payroll) test-taxes)

(define (compute-taxes payroll)
  (cond [(empty? payroll) . . . ]
        [(cons? payroll) (. . . (name (first payroll)) . . .
                          (amount (first payroll)) . . .
                          (compute-taxes (rest payroll)) . . . )]))

Now fill in the details...
One Version of `compute-taxes`

;; (compute-taxes payroll) calculates the tax owed for each
;; employee/salary pair in the payroll.
;; compute-taxes: Payroll → TaxOwed

(check-expect (compute-taxes test-payroll) test-taxes)

(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
    [(cons? payroll) (cons (list (name (first payroll))
                              (tax-payable (amount (first payroll))))
                        (compute-taxes (rest payroll)))]))

We could create a helper function to translate Payroll into TaxOwed.
Another version of `compute-taxes`

```
;; (sr→tr salary-rec) consumes a salary record and produces the
;; corresponding tax record

;; sr→tr: (list Str Num)  →  (list Str Num)
(define (sr→tr salary-rec)
  (list (name salary-rec) (tax-payable (amount salary-rec))))

(define (compute-taxes-alt payroll)
  (cond [(empty? payroll) empty]
        [(cons? payroll) (cons (sr→tr (first payroll))
                                (compute-taxes-alt (rest payroll)))]))
```
Alternate templates leading to the second solution

;; salary-rec-template: Payroll → Any
(define (salary-rec-template sr) (... (name sr) ... (amount sr) ...))

;; (payroll-template pr)

;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) (... (salary-rec-template (first pr)) ... ... (payroll-template (rest pr)) ...)]))
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. A Payroll is 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

Once upon a time, a dictionary was a book in which you look up a word to find a definition. Nowadays, a dictionary is an app:

`half of the words in his text were not in the dictionary: LEXICON, wordbook, glossary, vocabulary list, vocabulary, word list, wordfinder.`
More generally, a dictionary contains a number of **keys**, each with an associated **value**.

Examples:

- Your contacts list. Keys are names, and values are telephone numbers.
- Your seat assignment for midterms. Keys are userids, and values are seat locations.
- Stock symbols (keys) and prices (values).

Many two-column tables can be viewed as dictionaries. The previous examples can all be viewed as two-column tables.
Dictionary operations

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

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

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

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

;; An association list (AL) is one of:
;; ⋆ empty
;; ⋆ (cons (list Num Str) AL)
Association lists: an example

We can create association lists based on other types for keys and values. We use Nat and Str here just to provide a concrete example or an association list where given a student number you can look up the student’s name.

*Key Point:* 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 Nat Str))`. 
Association lists: template

We can use the data definition to produce a template.

;; al-template: AL → Any
(define (al-template alst)
  (cond [(empty? alst) . . . ]
        [else (. . . (first (first alst)) . . . ; first key
                    (second (first alst)) . . . ; first value
                    (al-template (rest alst)))]))
Association lists: lookup operation

Recall that lookup consumes a key and a dictionary (association list) and produces the corresponding value.

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

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

(check-expect (lookup 3 (list (list 1 "John") (list 3 "Pat"))) "Pat")
(check-expect (lookup 2 (list (list 1 "John") (list 3 "Pat"))) false)
Association lists: **lookup-al** implementation

;;; (lookup-al k alst) produces the value corresponding to key k,
;;; or false if k not present
;;; lookup-al:

(define (lookup-al k alst)
  (cond [(empty? alst) false]
        [(= k (first (first alst))) (second (first alst))]
        [else (lookup-al k (rest alst))])))
Association lists: further tasks

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

*Key Idea:* Another use of lists of lists is to represent a two-dimensional table (or a matrix).

For example, here is a multiplication table:

\[
\text{(mult-table 3 4)} \Rightarrow \\
\text{(list (list 0 0 0 0) (list 0 1 2 3) (list 0 2 4 6))}
\]

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

We can write `mult-table` using two applications of the “count up” idea.
Two-dimensional data example: part 1

;; (mult-table nr nc) produces multiplication table
;; with nr rows and nc columns
;; mult-table: Nat Nat → (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 → (listof (listof Nat))
(define (rows-from r nr nc)
  (cond [(>= r nr) empty]
        [else (cons (row r nc) (rows-from (add1 r) nr nc))])))
Two-dimensional data example: part 2

;; (row r nc) produces rth row of mult. table of length nc
;; row: Nat Nat → (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 → (listof Nat)
(define (cols-from c r nc)
  (cond [(≥ c nc) empty]
        [else (cons (* r c) (cols-from (add1 c) r nc))])))
Two-dimensional data - explained

- The parameters for the function `mult-table` specifies the number of rows (nr) and the number of columns (nc) in the table.

- `mult-table` uses `rows-from` to create a list of rows.

- `rows-from`: the parameter r goes from 0 up to nr and `rows` is used to create each row.

- `rows`: sets up the arguments for `cols-from`.

- `cols-from`: the parameter c goes from 0 up to nc and creates the row as a (listof Nat).
Processing two lists simultaneously

We now look at a more complicated recursion, namely 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.

1. processing just one list recursively
2. processing two lists in lockstep
3. processing two lists at different rates

The simplest case is when one of the lists does not require recursive processing.
Case 1: processing just one list: e.g. my-append

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) → (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)
  ...)

CS 135 Fall 2019 08: More lists 43
Case 1: **my-append** implementation

```scheme
(define (my-append lst1 lst2)
  (cond [(empty? lst1) lst2]
    [else (cons (first lst1)
                 (my-append (rest lst1) lst2))]))
```

The code only does simple recursion on \texttt{lst1}.

The parameter \texttt{lst2} is “along for the ride”.

\texttt{append} is a built-in function in Racket.
Case 1: A condensed trace of **my-append**

\[
\text{(my-append (cons 1 (cons 2 empty)) (cons 3 (cons 4 empty)))}
\]
\[
\Rightarrow \text{(cons 1 (my-append (cons 2 empty) (cons 3 (cons 4 empty))))}
\]
\[
\Rightarrow \text{(cons 1 (cons 2 (my-append empty (cons 3 (cons 4 empty))))})
\]
\[
\Rightarrow \text{(cons 1 (cons 2 (cons 3 (cons 4 empty))))}
\]
Case 2: processing in lockstep

In general, two lists can either be empty or a cons (four combinations in total).

*Key Idea:* To process two list in *lockstep*, they must be *the same length* and are *consumed at the same rate*.

Since 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.
Case 2: template for processing in lockstep

\[
\text{(define (lockstep-template lst1 lst2)}
\begin{align*}
\text{ & \quad (cond \[(empty? \text{lst1}) \ldots ]} \\
\text{ & \quad [else} \\
\text{ & \quad \quad (\ldots (first \text{lst1}) \ldots (first \text{lst2}) \ldots} \\
\text{ & \quad \quad (\text{lockstep-template (rest \text{lst1}) (rest \text{lst2}) \ldots )})])])
\end{align*}
\]
Case 2: dot product example

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`. 
Case 2: **dot-product** implementation

;; (dot-product lon1 lon2) computes the dot product
;; of vectors lon1 and lon2
;; dot-product: (listof Num) (listof Num) → 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)
  . . . )
Case 2: **dot-product** implementation

\[
\text{(define (dot-product lon1 lon2)}
\]

\[
\text{(cond [(empty? lon1) 0]}
\]

\[
\text{[else (+ (* (first lon1) (first lon2))}
\]

\[
\text{ (dot-product (rest lon1) (rest lon2))))])}
\]
Case 2: **dot-product** condensed trace

\[
\text{(dot-product (cons 2 (cons 3 empty))}
\]
\[
\text{(cons 4 (cons 5 empty)))}
\]
\[
\Rightarrow (+ 8 \text{(dot-product (cons 3 empty)}
\]
\[
\text{(cons 5 empty))}
\]
\[
\Rightarrow (+ 8 (+ 15 \text{(dot-product empty}
\]
\[
\text{empty)}))
\]
\[
\Rightarrow (+ 8 (+ 15 0)) \Rightarrow 23
\]
Case 3: processing at different rates

*Key Strategy:* If the two lists \( \text{lst1}, \text{lst2} \) being consumed are of different lengths, all four combinations of being empty/nonempty are possible:

\[
\text{(and (empty? \text{lst1}) (empty? \text{lst2}))}
\]
\[
\text{(and (empty? \text{lst1}) (cons? \text{lst2}))}
\]
\[
\text{(and (cons? \text{lst1}) (empty? \text{lst2}))}
\]
\[
\text{(and (cons? \text{lst1}) (cons? \text{lst2}))}
\]

Exactly one of these is true, but all must be tested in the template.
Case 3: the template so far

(define (twolist-template 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.
Case 3: refining the template

(define (twolist-template 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.
Case 3: further refinements

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

\[
\ldots (\text{first lst2}) \ldots (\text{twolist-template lst1 (rest lst2)}) \ldots \\
\ldots (\text{first lst1}) \ldots (\text{twolist-template (rest lst1) lst2}) \ldots \\
\ldots (\text{first lst1}) \ldots (\text{first lst2}) \ldots (\text{twolist-template (rest lst1) (rest lst2)}) \ldots
\]

We need to reason further in specific cases to determine which is appropriate.
Case 3 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:

\[
\text{(merge (list 1 8 10) (list 2 4 6 12)) } \Rightarrow \text{(list 1 2 4 6 8 10 12)}
\]

We need more examples to be confident of how to proceed.
Case 3 Example: more merging examples

(merge empty empty) ⇒ empty
(merge empty (list 2)) ⇒ (list 2)
(merge (list 1 3) empty) ⇒ (list 1 3)
(merge (list 1 4) (list 2)) ⇒ (list 1 2 4)
(merge (list 3 4) (list 2)) ⇒ (list 2 3 4)
Case 3 Example: 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).
Case 3 Example: implementation of \texttt{merge}

\begin{verbatim}
(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)))]))]
\end{verbatim}
Case 3 Example: a condensed trace

(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.

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

E.g. consider predicate “Does $e$ 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)?” produces `true`.
Examples for 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) → Bool

(check-expect (at-least? 0 'red (list 1 2 3)) true)
(check-expect (at-least? 3 "hi" empty) false)
(check-expect (at-least? 2 'red (list 'red 'blue 'red 'green)) true)
(check-expect (at-least? 3 'red (list 'red 'blue 'red 'green)) false)
(check-expect (at-least? 1 7 (list 5 4 0 5 3)) false)

(define (at-least? n elem lst) . . . )
Developing the code for **at-least?**

The recursion will involve the parameters \(n\) and \(lst\), once again giving four possibilities:

\[
\text{(define (at-least? n elem lst)} \\
\text{(cond [(and (zero? n) (empty? lst)) \ldots] \\
\text{[(and (zero? n) (cons? lst)) \ldots] \\
\text{[(and (> n 0) (empty? lst)) \ldots] \\
\text{[(and (> n 0) (cons? lst)) \ldots])])}
\]

Once again, exactly one of these four possibilities is true.
Refining \texttt{at-least}?

\begin{verbatim}
(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)) ???]))
\end{verbatim}

In which cases can we produce the answer without further processing?

In which cases do we need further recursive processing to discover the answer?
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 \( \text{lst} \) 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 \textit{at-least}?

\begin{verbatim}
(define (at-least? n elem lst)
  (cond [(zero? n) 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))])))
\end{verbatim}
Two condensed traces or \textbf{at-least}?

\[(\text{at-least? } 3 \text{'green} (\text{list 'red 'green 'blue})) \Rightarrow \]
\[(\text{at-least? } 3 \text{'green} (\text{list 'green 'blue})) \Rightarrow \]
\[(\text{at-least? } 2 \text{'green} (\text{list 'blue})) \Rightarrow \]
\[(\text{at-least? } 2 \text{'green empty}) \Rightarrow \text{false} \]

\[(\text{at-least? } 1 \text{ 8} (\text{list 4 8 15 16 23 42})) \Rightarrow \]
\[(\text{at-least? } 1 \text{ 8} (\text{list 8 15 16 23 42})) \Rightarrow \]
\[(\text{at-least? } 0 \text{ 8} (\text{list 15 16 23 42})) \Rightarrow \text{true} \]
Another example: testing list equality

;;; (list=? lst1 lst2) determines if lst1 and lst2 are equal
;;; list=?: (listof Num) (listof Num) → 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)) ??? ]))

Again there are four cases to consider.
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))
Implementation of list=?

(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 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 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.
Module 08 Summary

List Abbreviations and Dictionaries

1. The functions first, second,... eighth may be used to access elements at a particular location in a list. [12]
2. list creates a list whereas cons grows a list by one. [13]
3. Use ’ to create a list of Num, Sym, Str, Char or an empty list. [14]
4. You can create a list of lists. [15-18]
5. A flat list is a list that does not contain other lists. [29]
6. A nested list is a list which may contain lists which contain lists, and so on to an arbitrary depth. [29]
7. A dictionary associates a key with a value. [31]
Module 08 Summary

Dictionaries

8. Typical dictionary operations include [32]
   • **lookup** the value associated with a particular key,
   • **add** a key value pair to the dictionary
   • **remove** given a key, remove the key value pair.

9. An **association list** is a list of (key, value) pairs that contains at most one occurrence of any key. [33–34] 

10. Use lists of lists to represent a 2-D table or a matrix. [39–41]
Module 08 Summary

Processing Two Lists Simultaneously

11. When processing two lists simultaneously, there are three types of approaches

(a) processing just one list recursively [43–45]
(b) processing two lists in lockstep [46–51]
(c) processing two lists at different rates [52–60, 68-71]

12. The ideas behind processing two lists can be modified to process a list and a number. [61–67]
Patterns of recursion

Readings: none.

Topics:

• Simple recursion [3–6]
• Measuring Efficiency [7–8]
• Using accumulative recursion [9–16]
• Recognizing generative recursion [17-19]
Simple vs. general recursion

All of the recursion we have done to date has followed a pattern we call **simple recursion**.

The templates we have been using have been derived from a data definition and specify the form of the recursive application.

We will now learn to use a new pattern of recursion, **accumulative recursion**, and learn to recognize **generative recursion**.

**Approach:** For the next several lecture modules we will use simple recursion and accumulative recursion. We will avoid generative recursion until the end of the course.
Simple recursion

Recall from Module 06:

In simple recursion, every argument in a recursive function application (or applications, if there are more than one) are either:

• *unchanged*, or

• *one step closer to a base case* according to a data definition
The limits of simple recursion

;; (max-list lon) produces the maximum element of lon
;; max-list: (listof Num) → 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 limits of simple recursion

The code for max-list is correct.

But computing \(\text{max-list (countup-to 1 25)}\) is very slow.

Why?

The initial application is on a list of length 25.

Key Point: There are two recursive applications on the rest of this list, which is of length 24.

Each of those applications makes two recursive applications (which in turn make two recursive applications ...).
**The limits of simple recursion**

max-list can make up to \(2^n - 1\) recursive applications on a list of length \(n\).
Measuring efficiency (optional)

Key Idea: We can take the number of recursive applications as a rough measure of a function’s efficiency.

list-max can take up to $2^n - 1$ recursive applications.

length makes $n$ recursive applications on a list of length $n$.

length is clearly more efficient than this version of list-max.

We say that length’s efficiency is proportional to $n$ and max-list’s efficiency is proportional to $2^n$.

We express the former as $O(n)$ and the later as $O(2^n)$. 
Families of algorithms (Optional)

There are “families” of algorithms with similar efficiencies.

Examples, from most efficient to least:

<table>
<thead>
<tr>
<th>“Big-Oh”</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>no recursive calls: first</td>
</tr>
<tr>
<td>O(lg (n))</td>
<td>divide input in half: binary-search</td>
</tr>
<tr>
<td>O((n))</td>
<td>one recursive application for each item: length</td>
</tr>
<tr>
<td>O((n^2))</td>
<td>an O((n)) application for each item: insertion-sort</td>
</tr>
<tr>
<td>O((2^n))</td>
<td>two recursive applications for each item: max-list</td>
</tr>
</tbody>
</table>

Much more about “Big-Oh” notation and efficiency in later courses.
Accumulative recursion

Intuitively, we can find the maximum of a list of numbers by scanning it, remembering the largest value seen so far.

*Key Idea:* Pass down that largest value seen so far as a parameter called an **accumulator**.

This parameter accumulates the result of prior computation, and is used to compute the final answer that is produced in the base case.

This approach results in the code on the next slide.
Example of accumulative recursion

;; (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 → 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)]))

The accumulator is max-so-far.
The accumulator is returned in the base case.
(define (max-list2 lon)
  (max-list/acc (rest lon) (first lon)))

A helper function `max-list2` sets the initial value of the accumulator and the applies `max-list/acc`.

(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

Now even `(max-list2 (countup-to 1 2000))` is fast.
More about accumulative recursion

This technique is known as accumulative recursion.

*Key Observation* It is more difficult to develop and reason about such code, which is why simple 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.
Indicators of the accumulative recursion pattern

• All arguments to recursive function applications are:
  – unchanged, or
  – one step closer to a base case in the data definition, or
  – a partial answer (passed in an accumulator).

• The value(s) in the accumulator(s) are used in one or more base cases.

• The accumulatively recursive function usually has a wrapper function that sets the initial value of the accumulator(s).
Another accumulative example: reversing a list

Using simple recursion:

;;; my-reverse: (listof X) → (listof X)
(define (my-reverse lst)
  (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, $O(n^2)$. 
Reversing a list with an accumulator

(define (my-reverse lst) ; primary 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))]))

This algorithm is $O(n)$. 
A condensed trace

\[ (\text{my-reverse} \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ \text{empty})))) \]
\[ \Rightarrow (\text{my-rev/acc} \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ \text{empty})))) \ (\text{empty}) \]
\[ \Rightarrow (\text{my-rev/acc} \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ \text{empty}))) \ (\text{cons} \ 1 \ \text{empty}) \]
\[ \Rightarrow (\text{my-rev/acc} \ (\text{cons} \ 3 \ \text{empty})) \ (\text{cons} \ 2 \ (\text{cons} \ 1 \ \text{empty})) \]
\[ \Rightarrow (\text{my-rev/acc} \ \text{empty} \ (\text{cons} \ 3 \ (\text{cons} \ 2 \ (\text{cons} \ 1 \ \text{empty})))) \]
\[ \Rightarrow (\text{cons} \ 3 \ (\text{cons} \ 2 \ (\text{cons} \ 1 \ \text{empty})))) \]
Generative Recursion: GCD (will cover later)

In Math 135, you learn that the Euclidean algorithm for Greatest Common Divisor (GCD) can be derived from the following identity for \( m > 0 \):

\[
gcd(n, m) = gcd(m, n \mod m)
\]

We also have \( gcd(n, 0) = n \).

E.g. \( gcd(100, 85) \Rightarrow gcd(85, 15) \Rightarrow gcd(15, 10) \Rightarrow gcd(10, 5) \Rightarrow gcd(5, 0) \Rightarrow 5 \)

We can turn this reasoning directly into a Racket function.
Generative Recursion: GCD (will cover later)

;; (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm
;; euclid-gcd: Nat Nat → Nat
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))

This function does not use simple or accumulative recursion.

The parameter n becomes m and
the parameter m becomes (remainder n m).
Generative Recursion (will cover later)

*Key Point:* The arguments in the recursive application were *generated* by performing 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. Avoid generative recursion until then.
Simple vs. accumulative vs. generative recursion (will cover later)

In **simple 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 the data definition.

In **accumulative recursion**, parameters are as above, plus *parameters containing partial answers* used in the base case.

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 simple recursion, accumulative recursion, and generative recursion.

You should be able to write functions using simple and accumulative recursion.

You should know that some functions are much more efficient than others, that efficiency is expressed with “Big-Oh” notation, and that you’ll learn more about this in future courses.
Module 09 Summary

Types of Recursion

1. A limitation of simple recursion is that repeated applications of recursion within a function can cause it to run very slowly. [4–7]

2. In **Accumulative Recursion** the parameters are similar to that of simple recursion but with one or more parameters called *accumulators*. [9–11, 20]

3. **Accumulators** are parameters that accumulate (pass down) partial answers which are used in the base case. [9–11, 20]

4. For **Generative Recursion** there are no constraints on the parameters. [20]
Structures

Readings: HtDP, sections 6, 7.

- Avoid 6.2, 6.6, 6.7, 7.4.

Topics:

- Compound data [2–3]
- Example: posn structures [4–11]
- Defining & stepping structures [12–17]
- Data definition and analysis [18–27]
- Mixed data [28–36]
- Lists vs. structures [37–43]
Compound data

*Recall:* We have used short, fixed-length, lists for data that seems to always belong together. For example, in M08 we had a “payroll” with names and salaries:

```
(list (list "Asha" 50000))
  (list "Joseph" 100000)
  (list "Sami" 10000))
```

A name and salary always go together in this application.
Structures

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

*Key Point:* Structures permit the “bundling” of several related 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.
Example: posn structures

There are functions to create a structure (called constructors) and to produce components of a structure (called selectors).

- The constructor function make-posn, has the contract
  ```scheme
  ;; make-posn: Num Num → Posn
  ```

- selector functions posn-x and posn-y, have the contracts
  ```scheme
  ;; posn-x: Posn → Num
  ;; posn-y: Posn → Num
  ```

Analogy: The constructor function is similar to cons while the selector functions are similar to first and rest.
Example: posn structures

(define mypoint (make-posn 8 1))
(posn-x mypoint) ⇒ 8
(posn-y mypoint) ⇒ 1

Possible uses of posn:

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

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

That is, 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: distance in 2D

\[
distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}
\]
Example: distance in 2D

Functions can take `posn`’s as arguments.

;; (distance posn1 posn2) computes the Euclidean distance between posn1 and posn2
;; distance: Posn Posn → 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 can produce posns

Functions can produce posn’s.

;; (point-on-line slope intercept x) finds the point on the line
;; with given slope and intercept that has the given x-coordinate
;; point-on-line: Num Num Num → Posn
;; Example:
(check-expect (point-on-line 3 7 2)
  (make-posn 2 13))

(define (point-on-line slope intercept x)
  (make-posn x (+ (* x slope) intercept)))
Another example: scaling

*Task*: Multiply each component by the scale factor.

;; (scale v factor) scales vector v by the given factor
;; scale: Posn Num → Posn
;; Example:
(check-expect (scale (make-posn 3 4) 0.5) (make-posn 1.5 2))

(define (scale v factor)
  (make-posn (* factor (posn-x v)) (* factor (posn-y v))))
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.
Defining structures

*Key Point:* If \texttt{posn} wasn’t built in, we could define it using the Racket function

\begin{verbatim}
(define-struct posn (x y))
\end{verbatim}

The arguments to the \texttt{define-struct} special form are:

- a *structure name* (e.g. \texttt{posn}), and
- a *list of field names* in parentheses.

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

The expression `(define-struct posn (x y))` creates functions that can be used to process `posn`’s.

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

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

The special form

\(\texttt{(define-struct sname (fname1 \ldots fnamen))}\)

defines the structure type \texttt{sname} and automatically defines the following primitive functions:

- **Constructor:** \texttt{make-sname}
- **Selectors:** \texttt{sname-fname1 \ldots sname-fnamen}
- **Predicate:** \texttt{sname?}

\texttt{Sname} may be used in contracts.
Semantics of structures

The *substitution rule* for the $i$th selector is:

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

Finally, the *substitution rules* for the new predicate are:

$$(\text{sname}? \ (\text{make-sname} \ v_1 \ldots \ v_n)) \Rightarrow \text{true}$$

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

In these rules, we again 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)))) ⇒

(make-posn 2 (* 0.5 2)) ⇒

(make-posn 2 1)
Data definition and analysis

Suppose we want to represent information associated with songs.

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

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 SongInfo

The following code

(define-struct songinfo (performer title genre length))
;; An SongInfo is a (make-songinfo Str Str Sym Nat)

creates the following functions:

- constructor make-songinfo,
- selectors songinfo-performer, songinfo-title, songinfo-genre, songinfo-length, and
- type predicate songinfo?.
Templates and data-directed design

As we noted earlier, one of the main ideas of the HtDP textbook is that the form of a program often mirrors the form of the data.

We make use of that for structures as well. Recall:

- 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

Key Point: 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. I.e. it “unpacks” the data.

;; songinfo-template: SongInfo → Any

(define (songinfo-template info)
  (... (songinfo-performer info) ...
       (songinfo-title info) ...
       (songinfo-genre info) ...
       (songinfo-length info) ...))
Example: **update-genre**

;; (update-genre oldinfo newgenre) produces a new SongInfo
;; with the same information as oldinfo, except the genre
;; is replaced by newgenre
;; update-genre: SongInfo Sym → SongInfo
;; Example:

(check-expect
  (update-genre
    (make-songinfo "C.O.C." "Eye For An Eye" 'Folk 78)
   'Punk)
  (make-songinfo "C.O.C." "Eye For An Eye" 'Punk 78))
Example: update-genre

;; update-genre: SongInfo Sym → SongInfo

(define (update-genre oldinfo newgenre)
  (make-songinfo
   (songinfo-performer oldinfo)
   (songinfo-title oldinfo)
   newgenre
   (songinfo-length oldinfo)))

Key Point: We could easily have done this without a template, but using it pays off when designing more complicated functions.
Stepping \texttt{update-genre}

\begin{verbatim}
(define mysong (make-songinfo "U2" "Twilight" ’Rap 262))
(update-genre mysong ’Rock)
⇒ (update-genre
  (make-songinfo "U2" "Twilight" ’Rap 262) ’Rock)
⇒ (make-songinfo
  (songinfo-performer (make-songinfo "U2" "Twilight" ’Rap 262))
  (songinfo-title (make-songinfo "U2" "Twilight" ’Rap 262))
  ’Rock
  (songinfo-length (make-songinfo "U2" "Twilight" ’Rap 262)))
\end{verbatim}
Stepping an example (cont.)

⇒ (make-songinfo
   "U2"
   (songinfo-title (make-songinfo "U2" "Twilight" 'Rap 262))
   'Rock
   (songinfo-length (make-songinfo "U2" "Twilight" 'Rap 262)))
⇒ (make-songinfo
   "U2" "Twilight" 'Rock
   (songinfo-length (make-songinfo "U2" "Twilight" 'Rap 262)))
⇒ (make-songinfo "U2" "Twilight" 'Rock 262)
Design recipe for compound data

*Key Point:* 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

**Key Point:** Do the usual design recipe for every function.

**Purpose:** Same as before.

**Contract:** Can use both built-in 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.
Mixed data

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

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

*Key Point:* We can use these predicates to check aspects of contracts and to deal with data of mixed type.

Example: multimedia files
Example: multimedia files

First provide a data definition.

```
(define-struct movieinfo (director title genre duration))
;; A MovieInfo is a (make-movieinfo Str Str Sym Num )
;;
;; An mminfo is one of:
;; ⋆ a SongInfo
;; ⋆ a MovieInfo

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

Next provide a template...
The template for mminfo

*Key Point: 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.

`; mminfo-template: MmInfo → Any
(define (mminfo-template info)
  (cond [(songinfo? info)
         (... (songinfo-performer info) ... 
              (songinfo-title info) ... )]; two more fields
       [(movieinfo? info)
         (... (movieinfo-director info) ... )]); three more fields
mminfo example

(define favsong (make-songinfo "Beck" "Tropicalia"
   'Alternative 185))
(define favmovie (make-movieinfo "Orson Welles" "Citizen Kane"
   'Classic 119))

;; (mminfo-artist info) produces performer/director name from info
;; mminfo-artist: MmInfo → Str
;; Examples:
(check-expect (mminfo-artist favsong) "Beck")
(check-expect (mminfo-artist favmovie) "Orson Welles")
mminfo example

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

**Key Point:** 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 SongInfo 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 or the following notation can be used

;;; mminfo-artist: (anyof SongInfo MovieInfo) → Str

can be used.
Checked functions

*Key Point:* We can write a *safe version* of `make-posn`.

```scheme
;; safe-make-posn: Num Num → 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”.
Mixed data and object-oriented programming

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.
Lists versus structures

Recall: in M08 we wrote a Payroll data definition for processing tax withholdings:

;; A Payroll is one of:

;; * empty

;; * (cons (list Str Num) Payroll)

Example payroll:

(list (list "Asha" 50000)
   (list "Joseph" 100000)
   (list "Sami" 10000))
We also developed a corresponding template:

;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) . . . ]
    [(cons? pr) (. . . (first (first pr)) . . .
                     . . . (first (rest (first pr))) . . .
                     . . . (payroll-template (rest pr)) . . . )]
    . . . (first (rest (first pr))) . . .
    . . . (payroll-template (rest pr)) . . . ]))

*Key Point:* We recognized that two helper functions would make our code more readable, namely *name* and *amount*. 
;; (name lst) produces the first item from lst – the name.
(define (name lst) (first lst))

;; (amount lst) produces the second item from lst – the amount.
(define (amount lst) (first (rest lst)))

;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) . . .]
        [(cons? pr) (. . . (name (first pr)) . . .
                          . . . (amount (first pr)) . . .
                          . . . (payroll-template (rest pr)) . . . )]))
Payroll with structures

We can improve readability using structures.

```
(define-struct payroll (name amount))
;; A Payroll is a (make-payroll Str Num)

(define (payroll-template pr)
  (cond [(empty? pr) . . . ]
        [(cons? pr) (. . . (payroll-name (first pr)) . . .
                      . . . (payroll-amount (first pr)). . .
                      . . . (payroll-template (rest pr)))
           ])
```

When should each of these two approaches be used?
Why use lists containings lists?

Recall that the result of our payroll program was a list of taxes owed. Except for the name, the data definition is exactly the same as Payroll.

;; A TaxOwed is one of:
;; * empty
;; * (cons (list Str Num) TaxOwed)
Why use lists containings lists?
We could write a single function to extract the names from either:
;; (name tax-rec) extracts the first item (the name) from tax-rec
(define (name tax-rec) (first tax-rec))

;; (list-names lst) produces a list of names from the payroll or ...
;; list-names: (anyof Payroll TaxOwed) → (listof Str)
(check-expect (list-names payroll) (list "Asha" "Joseph" "Sami"))

(define (list-names lst)
  (cond [(empty? lst) empty]
        [(cons? lst) (cons (name (first lst))
                         (list-names (rest lst)))]))
Why use lists containing lists?

*If we use lists, a single function, name-list, will produce a list of names for both a Payroll or a TaxOwed.*

*If we use structures, we would require two different functions or extra complexity in the same function to distinguish which structure selector to use.*

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).

Structures automatically create the selector functions we needed to make the list-based code *more readable*.

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

Structures are provided in all mainstream programming languages.
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 the similar uses of structures and fixed-size lists, and be able to write functions that consume either type of data.
Module 10 Summary

Structures

1. **Structures** are used to store related data together. [3]

2. A **constructor** is a function used to create a structure. [4]

3. A **selector** is a function used to access the individual components of a structure. [4]

4. A **structure** is a value (i.e. it won’t be further simplified). [6]

5. When there is no operation defined for a particular type of value (e.g. Sym or Int), it causes a run-time error. [11]

6. The special form **define-struct** is used to create new types of structures. [12]
Module 10 Summary

7. The parts of the structure are call fields. [12]

8. The special form `define-struct` creates a constructor, selectors and a predicate for the newly defined structure. [13]

9. **Key Idea:** the form of a function often mirrors the form of the data so we create templates as a framework for a function in which we fill in the specifics. [20]

10. The template for a function, consumes a structure and selects every field in the structure. [21]

11. Use **data definitions** to specify the types that comprise a structure. [26]
Module 10 Summary

12. Create a Data Definition and Template for each new structure type. [26]

13. Then follow the Design Recipe for each function. [27]

14. For mixed data, use cond and predicates that check types when processing the data. [28, 30]

15. Use anyof to list the mixed types explicitly in a contract. [33]

16. You may use predicates and error to check a function’s argument types. [34]
Module 10 Summary

Lists vs. Structures

18. When we use *lists instead of structures* we can often reuse functions on related types. [41]

19. When we use *structures instead of lists* we can avoid some programming errors, make our code more readable, and can adapt the design recipe to give more guidance for writing functions. [43]
Trees

Readings: HtDP, sections 14, 15, 16.

Topics:

- Introductory examples and terminology [2–5]
- Binary trees [6–16]
- Binary search trees [17–27]
- Augmenting trees [28–48]
- Binary expression trees [49–57]
- General arithmetic expression trees [58–80]
- Nested lists [81–90]
Example: binary expression trees

The expression \(((2 \times 6) + (5 \times 2)) / (5 - 3)\) can be represented as a tree:
Example: evolution trees

Information related to the evolution of species can also be represented as a tree. This tree shows how species evolved to become new species.
Tree terminology

- root
- leaves
- parent
- child
- siblings
- internal nodes
- subtree
- label
Tree-terminology

• **root**: top node in the tree (only node without a parent)

• **parent**: next node on the direct path up to the root

• **child**: next node on the path down to a leaf

• **sibling**: another child of your parent

• **leaf**: a node without any children

• **internal node**: a node with at least one child
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)
Binary trees

A **binary tree** is a tree with *at most two children for each node*.

Binary trees are a fundamental part of computer science, independent of what language you use.

Binary arithmetic expression trees and evolution trees are both examples of binary trees.

We’ll start with the simplest possible binary tree. It could be used to store a set of natural numbers.
Draw a binary tree:

```
  5
 / \ 
1   6
```

Note: We will consistently use Nats in our binary trees, but it could be a symbol, string, struct, ...
Binary tree data definition

(define-struct node (key left right))

;; A Node is a (make-node Nat BT BT)

;; A binary tree (BT) is one of:

;; ★ empty

;; ★ Node

The node’s label is called “key” in anticipation of using binary trees to implement dictionaries.

What is the template?
Example functions on a binary tree

Let us fill in the template to make a simple function: count how many nodes in the BT have a key equal to \( k \) :

\[
\text{;; count-nodes: BT Nat } \rightarrow \text{ Nat}
\]

\[
\text{(define (count-nodes tree k)}
\]

\[
\text{(cond [(empty? tree) 0]}
\]

\[
\text{[else (+ (cond [(= k (node-key tree)) 1]}
\]

\[
\text{[else 0])}
\]

\[
\text{(count-nodes (node-left tree) k)}
\]

\[
\text{(count-nodes (node-right tree) k))])}
\]
Example functions on a binary tree

Add 1 to every key in a given tree:

;; increment: BT → BT
(define (increment tree)
  (cond
    [(empty? tree) empty]
    [else (make-node (add1 (node-key tree))
                     (increment (node-left tree))
                     (increment (node-right tree))))])
Searching binary trees

We are now ready to try to search our binary tree for a given key. It will produce true if it’s in the tree and false otherwise.

Our strategy:

• See if the root node contains the key we’re looking for. If so, produce true.

• Otherwise, recursively search in the left subtree and in the right subtree. If either recursive search finds the key, produce true. Otherwise, produce false.
Searching binary trees

Now we can fill in our BT template to write our search function:

;; search-bt: Nat BT → Bool
(define (search-bt k tree)
  (cond [(empty? tree) false]
        [(= k (node-key tree)) true]
        [else (or (search-bt k (node-left tree))
                    (search-bt k (node-right tree))))])

Is this more efficient than searching a list?
Find the path to a key

Write a function, search-bt-path, that searches for an item in the tree. As before, it will return false if the item is not found. However, if it is found search-bt-path will return a list of the symbols 'left and 'right indicating the path from the root to the item.
Find the path to a key

(check-expect (search-bt-path 0 empty) false)
(check-expect (search-bt-path 0 test-tree) false)
(check-expect (search-bt-path 6 test-tree) empty)
(check-expect (search-bt-path 9 test-tree) '(right right left))
(check-expect (search-bt-path 3 test-tree) '(left))
;; search-bt-path: Nat BT → (anyof false (listof Sym))
(define (search-bt-path k tree)
  (cond [(empty? tree) false]
        [(= k (node-key tree)) empty]
        [(list? (search-bt-path k (node-left tree)))
         (cons 'left (search-bt-path k (node-left tree)))]
        [(list? (search-bt-path k (node-right tree)))
         (cons 'right (search-bt-path k (node-right tree)))]
        [else false]]))

Double calls to search-bt-path which is inefficient.
Finding the path to a key

• For `search-bt-path` the base cases produces two different types
  1. the `Bool`, `false`, if the key has not been found or
  2. a `(listof Sym)` if the key has been found.

• If one of the recursive applications (`node-left` vs. `node-right`) yielded a list, the appropriate symbol ('left vs. 'right) is `cons`ed to the answer.

• There may be two recursive applications of `search-bt-path` if `list?` is true, which is expensive.

• This function can be written to avoid two recursive calls...
(define (search-bt-path k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) empty]
    [else (choose-path (search-bt-path k (node-left tree))
                     (search-bt-path k (node-right tree))))])

(define (choose-path path1 path2)
  (cond [(list? path1) (cons 'left path1)]
        [(list? path2) (cons 'right path2)]
        [else false])))
Binary search trees

We will now make one change that can make searching much more efficient. This change will create a tree structure known as a binary search tree (BST).

For any given collection of keys, there is more than one possible tree.

How the keys are placed in a tree can improve the running time of searching the tree when compared to searching the same items in a list.
A Binary Search Tree (BST) is one of:

- empty
- a Node

```
(define-struct node (key left right))
```

A Node is a (make-node Nat 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
  (make-node 1 empty empty)
  (make-node 6
    empty
    (make-node 14
      empty
      empty)))
A BST example

There can be several BSTs holding a particular set of keys.
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 $(\text{make-node } k \ l \ r)$, and $k$ equals $n$, then we have found it.

Otherwise it might be in either of the trees $l$, $r$.

**Key Point:** we can use the ordering property of the BST to avoid one recursive call...
Searching in a BST

If \( n < k \), then \( n \) must be in \( l \) if it is present at all, and we only need to recursively search in \( l \).

If \( n > k \), then \( n \) must be in \( r \) if it is present at all, and we only need to recursively search in \( r \).

Either way, we save one recursive function application.
Searching in a BST

;; (search-bst k t) produces true if k is in t; false otherwise.
;; search-bst: Nat BST → Bool

(define (search-bst k t)
  (cond [(empty? t) false]
        [(= k (node-key t)) true]
        [(< k (node-key t)) (search-bst k (node-left t))]
        [(> k (node-key t)) (search-bst k (node-right t))])))
Adding to a BST

How do we add a new key, \( k \), to a BST \( t \)?

If \( t \) is \texttt{empty}, then the result is a BST with only one node.

Otherwise \( t \) is of the form \( \texttt{(make-node n l r)} \).

If \( k = n \), the key is already in the tree and we can simply return \( t \).

If \( k < n \), then the new key 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.
Creating a BST from a list

How do we create a BST from a list of keys?

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 k lst), we add the key k to the BST created from the list lst. The first key is inserted last.

It is also possible to write a function that inserts keys in the opposite order.
Binary search trees in practice

*Limitation*: If the BST has all left subtrees empty, it looks and behaves like a sorted list, and the advantage is lost.

*Solution*: In later courses, you will see ways to keep a BST “balanced” so that “most” nodes have nonempty left and right children. We will also cover better ways to analyze the efficiency of algorithms and operations on data structures.
Augmenting trees

So far nodes have been `(define-struct node (key left right))`.

We can augment the node with additional data:

`(define-struct node (key val left right)).`

- The name `val` is arbitrary – choose any name you like.
- The type of `val` is also arbitrary: could be a number, string, structure, etc.
- You could augment with multiple values.
- The set of keys remains unique; could have duplicate values.
**BST dictionaries**

An **augmented BST** can serve as a dictionary that can perform significantly better than an association list.

Recall from Module 08 that a dictionary stores a set of (key, value) pairs, with at most one occurrence of any key. A dictionary supports **lookup**, **add**, and **remove** operations.

We implemented dictionaries using an association list, a list of two-element lists. Search could be inefficient for large lists.

We need to modify **node** to include the value associated with the key. **search** needs to return the associated value, if found.
(define-struct node (key val left right))

;; A binary search tree dictionary (BSTD) is either
;; empty or (make-node Nat Str BSTD BSTD)

;; (search-bst-dict k t) produces the value associated with k
;; if k is in t; false otherwise.

;; search-bst: Nat BSTD → anyof(Str false)

(define (search-bst-dict k t)
  (cond[(empty? t) false]
       [(= k (node-key t)) (node-val t)]
       [(< k (node-key t)) (search-bst-dict k (node-left t))]
       [(> k (node-key t)) (search-bst-dict k (node-right t))])))
(define test-tree (make-node 5 "Susan"
  (make-node 1 "Juan" empty empty)
  (make-node 14 "David"
    (make-node 6 "Lucy" empty empty))))

(check-expect (search-bst-dict 5 empty) false)
(check-expect (search-bst-dict 5 test-tree) "Susan")
(check-expect (search-bst-dict 6 test-tree) "Lucy")
(check-expect (search-bst-dict 2 test-tree) false)
Evolutionary trees are another kind of augmented tree.

**Evolutionary trees** are another kind of augmented tree.
Evolutionary trees are binary trees that show the evolutionary relationships between species. Biologists believe that all life on Earth is part of a single evolutionary tree, indicating common ancestry.

*Internal nodes* represent an *evolutionary event* when a common ancestor species split into two new species. Internal nodes are augmented with the common ancestor species name and an estimate of how long ago the evolutionary event took place (in millions of years).

*Leaves* represent a *most recent species*. They are augmented with a name and whether the species is endangered.
The fine print

We’ve simplified a lot...

• The correct terms are “phylogenetic tree” and “speciation event”. Nodes are often called “taxonomic units”. This is an active area of research; see Wikipedia on “phylogenetic tree”.

• Evolutionary trees are built with incomplete data and theories, so there could be many different evolution trees.

• Leaves could represent extinct species that died off before splitting. Hence the term “most recent species”.
Representing evolutionary trees

Internal nodes each have exactly *two children*. Each internal node has the name of the common ancestor species and the estimated date of the evolutionary event.

*Leaves* have names and endangerment status of the most recent species.

The *order* of children does not matter.

The structure of the tree is dictated by a hypothesis about evolution.
Data definitions for evolutionary trees

;; An EvoTree (Evolution Tree) is one of:
;; ⋆ a RSpecies (recent species)
;; ⋆ a EvoEvent (evolutionary event)

(define-struct rspecies (name endangered))

;; A RSpecies is a (make-rspecies Str Bool)
(define-struct evoevent (name age left right))

;; A EvoEvent is a (make-evoevent Str Num EvoTree EvoTree)

Note that the EvoEvent data definition uses a pair of EvoTrees.
Constructing the example evolutionary tree

(define-struct rspecies (name endangered))
(define-struct evoevent (name age left right))

(define human (make-rspecies "human" false))
(define chimp (make-rspecies "chimp" true))
(define rat (make-rspecies "rat" false))
(define crane (make-rspecies "crane" true))
(define chicken (make-rspecies "chicken" false))
(define worm (make-rspecies "worm" false))
(define fruit-fly (make-rspecies "fruit fly" false))
Constructing the example evolutionary tree (cont)

(define primate (make-evoevent "Primate" 5 human chimp))
(define mammal (make-evoevent "Mammal" 65 primate rat))
(define bird (make-evoevent "Bird" 100 crane chicken))
(define vertebrate
  (make-evoevent "Vertebrate" 320 mammal bird))
(define invertebrate
  (make-evoevent "Invertebrate" 530 worm fruit-fly))
(define animal
  (make-evoevent "Animal" 535 vertebrate invertebrate))
EvoTree template
Derive the EvoTree template from the data definition.

;; evotree-template: EvoTree → Any
(define (evotree-template t)
  (cond [(rspecies? t) (rspecies-template t)]
        [(evoevent? t) (evoevent-template t)]))

This is a straightforward implementation based on the data definition. It’s also a good strategy to take a complicated problem (dealing with an EvoTree) and decompose it into simpler problems (dealing with a RSpecies or an EvoEvent).

Functions for these two data definitions are on the next slide.
EvoTree template

;; rspecies-template: RSpecies → Any
(define (rspecies-template rs)
  (... (rspecies-name rs) ... 
       (rspecies-endangered rs) ... ))

;; evoevent-template: EvoEvent → Any
(define (evoevent-template ee)
  (... (evoevent-name ee) ... 
       (evoevent-age ee) ... 
       (evoevent-left ee) ... 
       (evoevent-right ee) ... ))
We know that (evoevent-left ee) and (evoevent-right ee) are EvoTrees, so apply the EvoTree-processing function to them.

;;; evoevent-template: EvoEvent → Any
(define (evoevent-template ee)
  (... (evoevent-name ee) ...)
  (evoevent-age ee) ...)
  (evotree-template (evoevent-left ee)) ...)
  (evotree-template (evoevent-right ee)) ...)))

Note; evoevent-template uses evotree-template and evotree-template uses evoevent-template. This arrangement is called mutual recursion.
A function on EvoTrees

This function counts the number of recent species within an EvoTree.

;; (count-species t): Counts the number of recent species
;;   (leaves) in the EvoTree t.
;; count-species: EvoTree  →  Nat
(define (count-species t)
  (cond [(rspecies? t) (count-recent t)]
        [(evoevent? t) (count-evoevent t)]))

(check-expect (count-species animal) 7)
(check-expect (count-species human) 1)
A function on EvoTrees (cont)

;; count-recent RSpecies → Nat
(define (count-recent t)
  1)

;; count-evoevent EvoEvent → Nat
(define (count-evoevent t)
  (+ (count-species (evoevent-left t))
      (count-species (evoevent-right t))))
A function on EvoTrees

- `count-species` checks whether the node is a leaf (`rspecies`) or an internal node (`evoevent`) and applies the appropriate function.

- `count-recent` returns 1 for each leaf (recent species)

- `count-evoevent` adds the values obtained from applying `count-species` on its two children.

- `count-species` applies `count-evoevent` and `count-evoevent` applies `count-species`. This pairing is an example of mutual recursion.
Traversing a tree

A **tree traversal** refers to the process of *visiting each node in a tree exactly once*. The increment example from binary trees is one example of a traversal.

We’ll traverse an EvoTree to produce a list of the names it contains.

We can solve this problem two different ways: using **append** or using accumulative recursion.

But first the format of the overall function **list-names** followed by the two functions for handling the two different types **EvoTrees**: **list-rs-names** and **list-ee-names**...
;; list-names: EvoTree → (listof Str)
(define (list-names t)
  (cond [(rspecies? t) (list-rs-names t)]
        [(evoevent? t) (list-ee-names t)]))

;; list-rs-names: RSpecies → (listof Str)
(define (list-rs-names rs)
  (... (rspecies-name rs) ...))

;; list-ca-names: EvoEvent → (listof Str)
(define (list-ee-names ee)
  (... (evoevent-name ee) ...
       (list-names (evoevent-left ee)) ...
       (list-names (evoevent-right ee)) ...))
list-names with an accumulator

;; list-names: EvoTree → (listof Str)
(define (list-names t)
  (list-names/acc t empty))

;; list-names/acc: EvoTree (listof Str) → (listof Str)
(define (list-names/acc t names)
  (cond [(rspecies? t) (list-rs-names t names)]
        [(evoevent? t) (list-ee-names t names)]))
;; list-rs-names: RSpecies (listof Str) → (listof Str)
(define (list-rs-names rs names)
  (cons (rspecies-name rs) names))

;; list-ee-names: EvoEvent (listof Str) → (listof Str)
(define (list-ee-names ee names)
  (cons (evoevent-name ee)
        (list-names/acc (evoevent-left ee)
                        (list-names/acc (evoevent-right ee) names)))))

(check-expect (list-names human) '("human"))
(check-expect (list-names mammal)
             '("Mammal" "Primate" "human" "chimp" "rat"))
Practice problems with EvoTrees

• Count the number of evolutionary events (internal nodes) with
and age less than n. For example, the sample tree has 4 events
that are less than 400 million years old.

• Count the number of evolutionary events that occurred to
produce a given recent species.

• Find the evolutionary path between the root of a (sub)tree and a
recent species. For example, the path from animal to rat is
'(animal vertebrate mammal rat).

• Modify list-names to produce the names of endangered species.
Binary expression trees

The arithmetic expression \(((2 \times 6) + (5 \times 2))/(5 - 3)\) can be represented as a tree:
Representing binary arithmetic expressions

*Internal nodes* each have exactly two children.

*Internal nodes* have symbol labels.

*Leaves* have number labels.

We care about *the order* of children.

The structure of the tree is dictated by the expression.
(define-struct binode (op left right))
;; 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

Some examples of binary arithmetic expressions:

5
(make-binode ' ∗ 2 6)
(make-binode ' + 2 (make-binode ' − 5 3))
A more complex example:

\[
\text{(make-binode '/}
(\text{make-binode '+ (make-binode ' ∗ 2 6)}
(\text{make-binode ' ∗ 5 2}))
(\text{make-binode ' − 5 3}))
\]
Templates for binary arithmetic expressions

;; binexp-template: BinExp → Any
(define (binexp-template ex)
  (cond [(number? ex) (... ex ...)]
        [(binode? ex) (binode-template ex)]))

;; binode-template: BINode → Any
(define (binode-template node)
  (... (binode-op node) ...
       (binexp-template (binode-left node)) ...
       (binexp-template (binode-right node)) ...)))
Evaluating expressions

;; (eval ex) evaluates the expression ex and produces its value.
;; eval: BinExp → Num

(check-expect (eval 5) 5)
(check-expect (eval (make-binode '+ 2 5)) 7)
(check-expect (eval (make-binode '/ (make-binode '-' 10 2)
                        (make-binode '+ 2 2))) 2)

(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode ex)]))
(define (eval-binode node)
  (cond [(symbol = ? '∗ (binode-op node))
         (∗ (eval (binode-left node)) (eval (binode-right node)))]
       [(symbol = ? '/ (binode-op node))
        (/ (eval (binode-left node)) (eval (binode-right node)))]
       [(symbol = ? ' + (binode-op node))
        (+ (eval (binode-left node)) (eval (binode-right node)))]
       [(symbol = ? ' − (binode-op node))
        (− (eval (binode-left node)) (eval (binode-right node)))])))
Eval, refactored

(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode (binode-op ex)
                                (eval (binode-left ex))
                                (eval (binode-right ex)))]))

(define (eval-binode op left right)
  (cond [(symbol = ? op '∗) (∗ left right)]
        [(symbol = ? op '/) (/ left right)]
        [(symbol = ? op ’+’) (+ left right)]
        [(symbol = ? op ’−’) (− left right)]))
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?

*Key Question:* What if there can be any number of children?
General arithmetic expressions

For binary arithmetic expressions, we formed binary trees.

*Key Observations:* Racket expressions using the functions $+$ and $\ast$ can have an unbounded number of arguments.

For simplicity, we will restrict the operations to $+$ and $\ast$.

$$(+ (\ast 4 2) 3 (+ 5 1 2) 2)$$
Visualizing an arithmetic expression

We can visualize an arithmetic expression as a general tree.

\((+ (* 4 2) 3 (+ 5 1 2) 2)\)
General arithmetic expressions

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).
(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. Examples: An EvoTree was defined in terms of RSpecies and EvoEvent. EvoEvent was defined in terms of EvoTree. BINode and BinExp depend on each other.
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))

(make-ainode '+ (list))
Templates for arithmetic expressions

;; aexp-template: AExp → Any
(define (aexp-template ex)
  (cond [(number? ex) (. . . ex . . . )]
        [(ainode? ex) (. . . (ainode-op ex) (. . . (aexp-template (first args)) . . .
                                           (listof-aexp-template (rest args)) . . . )
           (listof-aexp-template (ainode-args ex)))])
)

;; listof-aexp-template: (listof AExp) → Any
(define (listof-aexp-template args)
  (cond [(empty? args) . . . ]
        [else (. . . (aexp-template (first args)) . . .
                (listof-aexp-template (rest args)) . . . )])
)
The function eval

;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num

(check-expect (eval 3) 3)
(check-expect (eval (make-ainode '+ (list 3 4))) 7)
(check-expect (eval (make-ainode '+ ())) 0)

(define (eval ex)
    (cond [(number? ex) ex]
          [(ainode? ex) (apply (ainode-op ex) (ainode-args ex))])))
;; (apply op exlist) applies op to the list of arguments.
;; apply: op (listof AExp) → Num
(define (apply op args)
  (cond [(empty? args) (cond [(symbol=? op '+) 0]
                               [(symbol=? op '* ) 1])]
     [(symbol=? op '+) (+ (eval (first args))
                          (apply op (rest args)))]
     [(symbol=? op '* ) (* (eval (first args))
                           (apply op (rest args)))]))
Condensed trace of aexp evaluation

\[
(\text{eval } (\text{make-ainode } '+ (\text{list } (\text{make-ainode } ' \times ' '(3 4)) \\
\quad (\text{make-ainode } ' \times ' '(2 5))))) \\
\Rightarrow (\text{apply } '+ (\text{list } (\text{make-ainode } ' \times ' '(3 4)) \\
\quad (\text{make-ainode } ' \times ' '(2 5))))) \\
\Rightarrow (+ (\text{eval } (\text{make-ainode } ' \times ' '(3 4)) \\
\quad (\text{apply } '+ (\text{list } (\text{make-ainode } ' \times ' '(2 5))))) \\
\Rightarrow (+ (\text{apply } ' \times ' '(3 4)) \\
\quad (\text{apply } '+ (\text{list } (\text{make-ainode } ' \times ' '(2 5)))))
\]
\[ \Rightarrow (\text{+} (\times (\text{eval} \ 3) (\text{apply} \ \times \ (4)))
\]
\[ (\text{apply} \ \text{+} (\text{list} (\text{make-ainode} \ \times \ (2 \ 5)))))) \]
\[ \Rightarrow (\text{+} (\times 3 (\text{apply} \ \times \ (4)))
\]
\[ (\text{apply} \ \text{+} (\text{list} (\text{make-ainode} \ \times \ (2 \ 5)))))) \]
\[ \Rightarrow (\text{+} (\times 3 (\times (\text{eval} \ 4) (\text{apply} \ \times \ \text{empty})))
\]
\[ (\text{apply} \ \text{+} (\text{list} (\text{make-ainode} \ \times \ (2 \ 5)))))) \]
\[ \Rightarrow (\text{+} (\times 3 (\times 4 (\text{apply} \ \times \ \text{empty})))
\]
\[ (\text{apply} \ \text{+} (\text{list} (\text{make-ainode} \ \times \ (2 \ 5)))))) \]
⇒ (+ (* 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)))
⇒ (+ 12 (+ (* 2 (apply '∗ '(5)))) (apply ' + empty)))
⇒ (+ 12 (+ (* 2 (* (eval 5) (apply '∗ empty)))) (apply ' + empty)))
⇒ (+ 12 (+ (* 2 (* 5 (apply '∗ empty)))) (apply ' + empty)))
⇒ (+ 12 (+ (* 2 (* 5 1)) (apply ' + empty)))
⇒ (+ 12 (+ (* 2 5) (apply ' + empty)))
⇒ (+ 12 (+ 10 (apply ' + empty)))
⇒ (+ 12 (+ 10 0)) ⇒ (+ 12 10) ⇒ 22
Alternate data definition *Not covered this term!*

In Module 8, we saw how a list could be used instead of a structure holding tax record information.

Here we could use a similar idea to replace the structure `ainode` and the data definitions for `AExp`. 
Not covered this term!

;; An alternate arithmetic expression (AltAExp) is one of:
;; ⋆ a Num
;; ⋆ (cons (anyof '∗ ' + ... operation) and a
list of expressions.
3
'( + 3 4)
'( + ( ∗ 4 2 3) ( + ( ∗ 5 1 2) 2))

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) Not covered this term!

\[
\text{(define (altaexp-template ex))}
\]

\[
\text{(cond [(number? ex) (\ldots ex \ldots)]}
\]

\[
[\text{else (\ldots (first ex) \ldots )}}
\]

\[
\text{(listof-altaexp-template (rest ex)) \ldots ]})]]
\]

\[
\text{(define (listof-altaexp-template exlist))}
\]

\[
\text{(cond [(empty? exlist) \ldots ]}
\]

\[
[(\text{cons? exlist) (\ldots (altaexp-template (first exlist)) \ldots}
\]

\[
\text{(listof-altaexp-template (rest exlist)) \ldots ]})])
\]
Not covered this term!

;; eval: AltAExp → Num

(define (eval aax)
  (cond [(number? aax) aax]
        [else (apply (first aax) (rest aax))])))
Not covered this term!

;; apply: Sym (listof AltAExp) → 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 \textit{Not covered this term!}

\begin{align*}
\text{(eval } & (\ast (\, (+ \, 1 \, 2) \, 4)) \\
\Rightarrow & \text{(apply } \ast \, ((+ \, 1 \, 2) \, 4)) \\
\Rightarrow & (\ast (\text{eval } (+ \, 1 \, 2)) \text{ (apply } \ast \, (4)))) \\
\Rightarrow & (\ast (\text{apply } + \, (1 \, 2)) \text{ (apply } \ast \, (4)))) \\
\Rightarrow & (\ast (\, + \, 1 \, (\text{apply } + \, (2))) \text{ (apply } \ast \, (4)))) \\
\Rightarrow & (\ast (\, + \, 1 \, (\, + \, 2 \, (\text{apply } + \, ())) \text{ (apply } \ast \, (4)))) \\
\Rightarrow & (\ast (\, + \, 1 \, (\, + \, 2 \, 0) \text{ (apply } \ast \, (4))))
\end{align*}
Not covered this term!

⇒ (∗ (+ 1 2) (apply ’∗ ’(4)))
⇒ (∗ 3 (apply ’∗ ’(4)))
⇒ (∗ 3 (∗ 4 (apply ’∗ ’())))
⇒ (∗ 3 (∗ 4 1))
⇒ (∗ 3 4)
⇒ 12
Structuring data using mutual recursion

*Key Point:* 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.
Text example

'(chapter
  (section
    (paragraph "This is the first sentence."
       "This is the second sentence."
       "We can continue in this manner."
    )
    (paragraph "We can continue in this manner."
    )
  )
  (section . . .
    ...
    )
)

Web page example

'(webpage
   (title "CS 135: Designing Functional Programs")
   (paragraph "For a course description,"
      (link "click here." "desc.html")
      "Enjoy the course!"
   )
   (horizontal-line)
   (paragraph "(Last modified yesterday.)")
)
Nested lists

We have discussed flat lists (no nesting):

'( a 1 "hello" x)

and lists of lists (one level of nesting):

'((1 "a") (2 "b"))

We now consider nested lists (i.e. there can be arbitrary levels of nesting).

E.g.

'(((1 (2 3)) 4 (5 (6 7 8) 9))
It is often useful to visualize a nested list as a tree, in which the leaves correspond to the elements of the list, and the internal nodes indicate the nesting:

Which is a tree version of the following list.

```
'((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)

This can be generalized to generic types: (Nest-List-X)

Since there are three cases in our data definition, we would expect a
(cond ... ) with three cases.
Template for nested lists

The template follows from the data definition.

\[
\text{nest-lst-template: Nest-List-Num} \rightarrow \text{Any}
\]

\[
\text{(define (nest-lst-template lst)}
\]

\[
\text{(cond [(empty? lst) \ldots ]}
\]

\[
\text{[(number? (first lst))}
\]

\[
\text{(\ldots (first lst) \ldots \text{(nest-lst-template (rest lst)}) \ldots ]})
\]

\[
\text{[else}
\]

\[
\text{(\ldots (nest-lst-template (first lst)) \ldots}
\]

\[
\text{(nest-lst-template (rest lst)) \ldots ]})
\]
Example: the function count-items

;; count-items: Nest-List-Num → Nat
(define (count-items nln)
  (cond 
    [(empty? nln) 0]
    [(number? (first nln)) ; it is a number
      (+ 1 (count-items (rest nln)))]
    [else (+ (count-items (first nln)) ; it is a list
             (count-items (rest nln)))]))
Condensed trace of count-items

(count-items '((10 20) 30))
⇒ (+ (count-items '(10 20)) (count-items '(30)))
⇒ (+ (+ 1 (count-items '(20))) (count-items '(30)))
⇒ (+ (+ 1 (+ 1 (count-items '()))) (count-items '(30)))
⇒ (+ (+ 1 (+ 1 0)) (count-items '(30)))
⇒ (+ (+ 1 1) (count-items '(30)))
⇒ (+ 2 (count-items '(30)))
⇒ (+ 2 (+ 1 (count-items '())))
⇒ (+ 2 (+ 1 0)) ⇒ (+ 2 1) ⇒ 3
Flattening a nested list

`flatten` produces a flat list from a nested list.

;; (flatten lst) produces a single-level list with all the elements of lst.

;; flatten: Nest-List-Num → (listof Num)

(check-expect (flatten '(1 2 3)) '(1 2 3))
(check-expect (flatten '((1 2 3) (a b c))) '(1 2 3 a b c))

(define (flatten lst) ... )

We make use of the built-in Racket function `append`.

(append '(1 2) '(3 4)) ⇒ '(1 2 3 4)
Flattening a nested list: implementation

;;; flatten: Nest-List-Num → (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 trees, binary search trees, evolutionary trees, and binary arithmetic expressions.

You should 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.
Module 11 Summary

Overview of Trees

1. In this module we introduce increasing sophisticated trees: a tree [2-5], a binary tree [6-16], a binary search tree [17-27], an augmented tree [28] (including a BST dictionary [29-31]), an evolutionary tree [32-48], a binary expression tree [49-56], general trees [57, 77-80], arithmetic expressions [58-69] and finally nested lists [81-90].

2. We skipped the topic “an alternative data definition for an arithmetic expression” [70-76].
Module 11 Summary

Trees, BSTs and BST Dictionaries

3. Terms use to describe a tree include root, parent, child, sibling, leaf, internal node and subtree. [4]

4. For a binary tree (BT) each internal node has at most two children, typically called left and right. [6]

5. A Node is a (define-struct node (key left right)). [8].

6. With a (listof X) there is a single recursive application to the rest of the list. With a binary tree there are two recursive applications: one to the left subtree and one to the right. [9]
Module 11 Summary

Search Trees and Dictionaries

7. A Binary Search Tree (BST) is a binary tree that has an ordering property: namely for each node, its key is greater than every key in its left subtree and is less than every key in its right subtree. [18]

8. An augmented tree is a tree that keeps additional information besides the key. [28]

9. A BST Dictionary (BSTD) is an Augmented BST where each node stores a key value pair and the operations lookup, add and remove are supported. [29–31]
Module 11 Summary
EvoTrees and Traversals

10. An Evolutionary tree (EvoTree) is an augmented tree that keeps information about the evolutionary relationship between species. [33]

11. Internal nodes represent one type of information, evolutionary events whereas the leaves represent another, most recent species. [33]

12. Mutual recursion is when two functions call each other. [41]

13. A tree traversal is the process of visiting each node in a tree exactly once. [44]
Module 11 Summary
Expression Trees and Nested Lists

14. A **binary expression tree** is a tree where the internal nodes are arithmetic operations (+, -, *, /) and the leaves are numbers. [50]

15. An **arithmetic expression** can be represented as a tree where the internal nodes can have a list of, i.e. more than two, children. [57-60]

16. **Nested lists** can also be represented as a tree where the leaves correspond to elements in the list. [82-85]
Local definitions and lexical scope

Readings: HtDP, Intermezzo 3 (Section 18).

Language level: Intermediate Student

Topics:

• Motivating local definitions [1–11]
• Semantics of local [12–16]
• Reasons to use local [17–42]
• Terminology [43–44]
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.

\[
\text{local \([(define \ x1 \ exp1) \ldots \ (define \ xn \ expn)\] bodyexp)}
\]

What use is this feature?
Motivating local definitions

Consider Heron’s formula for the area of a triangle with sides $a, b, c$:
$$\sqrt{s(s - a)(s - b)(s - c)}, \text{ where } s = (a + b + c)/2.$$ 

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.
Heron’s formula version 1

(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))))

*Key Point* The repeated computation of $s = (a + b + c)/2$ is awkward.
Heron’s formula version 2
We could notice that \( s - a = \frac{(-a + b + c)}{2} \), and make similar substitutions.

\[
\text{(define (t-area a b c)}
\begin{align*}
\text{ (sqrt} \\
\text{ (* (/(+ a b c) 2) } \\
\text{ (/ (+ (- a) b c) 2)) } \\
\text{ (/ (+ a (- b) c) 2)) } \\
\text{ (/ (+ a b (- c)) 2))))
\end{align*}
\]

This approach is shorter, but its relationship to Heron’s formula is unclear from just reading the code, and the technique does not generalize.
Heron’s formula version 3

*Key Idea:* use a helper function instead.

```
(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))
```
Pros and cons of using a helper function

_Progs:_ This generalizes well to formulas that define several intermediate quantities.

_Cons:_ But the helper functions need parameters, which again makes the relationship to Heron’s formula hard to see. And there’s still repeated code and repeated computations.
Heron’s formula version 4

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.

\[
\text{(define (t-area3/s a b c s)}
\text{  (sqrt (* s (- s a) (- s b) (- s c)))))}
\]
\[
\text{(define (t-area3 a b c)}
\text{  (t-area3/s a b c (/ (+ a b c) 2)))}
\]

**Pros:** This version is more readable and shorter.

**Cons:** But this version is still awkward: The value of \( s \) is defined in one function and used in another.
Heron’s formula using local

**Key Benefit:** 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)))))
```

**Convention:** Since local is another form (like cond) that results in double parentheses, we will use square brackets to improve readability. This is another convention.
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.

*Recall:* The substitution specified in the semantics of function application ensures that the correct value is used while evaluating the last line.
Reusing names

Key Point: Both function parameters and local have similar semantics.

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.

How it works: 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.
Semantics of local

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.
Example: evaluating t-area4

(t-area4 3 4 5) ⇒

(local [(define s (/ (+ 3 4 5) 2))]
  (sqrt (* s (- s 3) (- s 4) (- s 5))) ⇒

(define s_47 (/ (+ 3 4 5) 2))

(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5))) ⇒

(define s_47 (/ 12 2))

(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5))) ⇒

(define s_47 6)

(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5))) ⇒ ... 6
Semantics of `local`

In general, an expression of the form

```scheme
(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.
Semantics of `local`

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.
Reasons to use local

1. Clarity: Naming subexpressions [18–19]
2. Efficiency: Avoid recomputation [20–27]
3. Encapsulation: Hiding stuff [28–34]
4. Scope: Reusing parameters [35–42]
Upcoming Slides

The slides for much of the rest of the module take previously discussed functions and rewrite them using local.

- **max-list** (slides 21) is from module 09 slide 4
- **search-bt-path** (slide 23) is from module 11 slide 15
- **isort** (slide 32) is from module 08 slides 4, 8
- **countup-to** (slides 35–36) is from module 07 slides 21–22
- **mult-table** (slides 43–47) is from module 06 slides 39–40
1. Clarity: naming subexpressions

A subexpression used twice within a function body always yields the same value.

*Key Reason:* Using `local` to *give the reused subexpression a name improves the readability* of the code.

This was a motivating factor in `t-area`. Naming the subexpression made the relationship to Herron’s Formula clear.

```
(define (t-area4 a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c))))))
```
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.

```
(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))))))
```
2. Efficiency: avoid recomputation

Recall that in lecture module 09, we saw a version of max-list used the same recursive application twice. The repeated computation of values caused it to be very slow, even for lists of length 25.

*Key Reason:* We can use `local` to *avoid recomputation.*
Old version of **max-list**

;; (max-list lon) produces the maximum element of lon
;; max-list: (listof Num) → 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))])))
Improved version of \texttt{max-list}

\begin{verbatim}
;; 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]))]))
\end{verbatim}
Old version of `search-bt-path`

;; search-bt-path-v1: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v1 k tree) ) ; v1 original version

(cond
  [(empty? tree) false]
  [(= k (node-key tree)) '()]
  [(list? (search-bt-path-v1 k (node-left tree)))
   (cons 'left (search-bt-path-v1 k (node-left tree)))]
  [(list? (search-bt-path-v1 k (node-right tree)))
   (cons 'right (search-bt-path-v1 k (node-right tree)))]
  [else false]])
(define (search-bt-path-v2 k tree) ; Use a helper function
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]
    [else (choose-path (search-bt-path-v2 k (node-left tree))
                        (search-bt-path-v2 k (node-right tree)))]
  )
)

(define (choose-path path1 path2)
  (cond [(list? path1) (cons 'left path1)]
        [(list? path2) (cons 'right path2)]
        [else false]))
Using **local**: 

;;; search-bt-path-v3: Nat BT → (anyof false (listof Sym))

(define (search-bt-path-v3 k tree) ; use local
  (cond 
    [(empty? tree) false]
    [(= k (node-key tree)) ’()]
    [else (local [(define left (search-bt-path-v3 k (node-left tree)))
                  (define right (search-bt-path-v3 k (node-right tree)))]
               (cond [(list? left) (cons ’left left)]
                     [(list? right) (cons ’right right)]
                     [else false]))]))
Using **local**: 

Version 3 of **search-bt-path** 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 binary 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**.
;; search-bt-path-v4: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v4 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]
    [else (local [(define left (search-bt-path-v4 k (node-left tree)))]
      (cond [(list? left) (cons 'left left)]
        [else (local [(define right
          (search-bt-path-v4 k (node-right tree)))]
          (cond [(list? right) (cons 'right right)]
            [else false]))]))]))
3. Encapsulation: hiding stuff

**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 246 we will see how objects combine data encapsulation with another type of encapsulation we now discuss.
Behaviour encapsulation

*Key Point:* 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 type of encapsulation is known as *behaviour encapsulation.*
Behaviour encapsulation: benefits

Behaviour encapsulation allows us to move helper functions within the function that uses them, so they:

- are *invisible outside* the function.
- *do not clutter* the “namespace” at the top level.
- *cannot be used by mistake*.

This makes the organization of the program more obvious and is particularly useful when using accumulators.
Example: **sum-list**

```
(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 reasoning with **invariants**. It will be discussed further in CS 245. It is important in CS 240 and CS 341.
Example: Insertion sort \texttt{isort}

\begin{verbatim}
(define (isort lon)
  (local [(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)))]))
\end{verbatim}

Here the \texttt{insert} helper function is included in the body of \texttt{isort}.
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) → (listof Num)
    ;; requires: slon is sorted in nondecreasing order
    (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)))]])])
4. Scope: reusing parameters

Making helper functions local can reduce the need to have parameters “go along for the ride”.

\[(\text{define (countup-to } n)\]
\[(\text{(countup-to-from } n \ 0))\]

\[(\text{define (countup-to-from } n \ m)\]
\[\text{(cond [(> } m \ n) \ \text{empty}]}\]
\[\text{[else (cons } m \ (\text{countup-to-from} \ n \ (\text{add1 } m)))]})\]

Here \(n\) must be a parameter in \text{countup-to-from} to know when to stop.
(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.
The semantics of local

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.

Multiple copies of very similar functions will exist.
Example **mult-table**

We can use the same idea to localize the helper functions for **mult-table** from lecture module 08.

Recall that

\[
\text{(mult-table 3 4) } \Rightarrow \\
\text{(list (list 0 0 0 0) (list 0 1 2 3) (list 0 2 4 6))}
\]

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

The next two slides present the code you’ve seen in Module 8.
mult-table explained

In the original version (slides 39–40)

- mult-table applies rows-from to produce the rows. rows-from applies rows to produce a single row. rows applies cols-from to produce each individual column entry in a row.

In the version using local (slide 41)

- rows-from ands row are functions local to mult-table2.
- row has its own local function cols-from.
;; code from lecture module 08
;; (mult-table nr nc) produces multiplication table
;; with nr rows and nc columns
;; mult-table: Nat Nat → (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 → (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 → (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 → (listof Nat)
(define (cols-from c r nc)
  (cond [(≥ c nc) empty]
        [else (cons (∗ r c) (cols-from (add1 c) r nc))))))
Example: mult-table using local

(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))
More on `mult-table2`

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 module 13.
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.*
Terminology associated with local

Looking back at slide 36...

- The binding occurrence of `countup2-to` and `n` occur on line 1.
- The binding occurrence of `countup-from` and `m` occur on line 3.
- The associated bound occurrence of `n` is on line 4 and `countup-from` is on lines 5 and 6.
- The lexical scope of the parameter `n` is the entire body of the `countup-to`.
- The lexical scope of the parameter `m` is within `local`.
- The function `countup2-to` has global scope.
Use of `local`

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.
Module 12 Summary

The Special Function `local`

1. The special form `local` contains a series of local definitions plus an expression using them. [2]

2. `local` can be used to avoid repetition of common subexpressions. [3-9]

3. `local` can reuse an existing name. [10-11]

4. Expressions in `local` are replaced with a fresh identifier and lifted to the top level in one step. [15]

5. Using `local` to name subexpressions can help with readability. [18]
Module 12 Summary

Encapsulation

6. **Encapsulation** is the idea of keeping related items together. [28]

7. Encapsulation can also help prevent unintended users from using the location definitions. [28]

8. Both values (**data encapsulation**) and functions (**behaviour encapsulation**) can be encapsulated. [28-29]

9. An **invariant** is a relationship that does not change during the course of the computation. [31]
Module 12 Summary

Binding and Scope

10. The binding occurrence of a name is its use in a definition or formal parameter to a function. [43]

11. The associated bound occurrences are the uses of that name that correspond to that binding. [43]

12. The lexical scope of a binding occurrence is all places where that binding has effect. [43]

13. Global scope is the scope of top-level definitions. [43]
Functional abstraction


Language level: Intermediate Student With Lambda

Topics:

• Functions are first class values
• Contracts and types
• Anonymous functions
• Syntax & semantics
• Abstracting from examples
• Higher-order functions
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

**Key Point:** 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.
Functions as first-class values

In the Intermediate language, functions are values. In fact, functions 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
Functions as first-class values has historically been missing from languages that are not primarily functional.

The utility of functions-as-values is now widely recognized, and they are at least partially supported in many languages that are not primarily functional, including C++, C#, Java, Go, JavaScript, Python, and Ruby.

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) ⇒ 5
(foo ∗ 2 3) ⇒ 6

The expression (foo + 2 3) is not only passing in 2 and 3 as arguments to foo, it is also passing in the function + as an argument.
(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))]))

If pred? is true then cons it to the answer. Otherwise skip that element and apply pred? to the rest of the list. I.e. keep the items in the list where pred? produces true.
Tracing my-filter

(my-filter odd? (list 5 6 7))
  ⇒ (cons 5 (my-filter odd? (list 6 7)))
  ⇒ (cons 5 (my-filter odd? (list 7)))
  ⇒ (cons 5 (cons 7 (my-filter odd? empty)))
  ⇒ (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.

More specifically functional abstraction is the process of combining two or more related functions into a single definition.

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 \texttt{local} could be used to create functions during a computation, to be used in evaluating the body of the \texttt{local}.

\textit{Key Point:} But now, because functions are values, the body of the \texttt{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) ⇒

((local [(define (f m) ( + 3 m)) f) 4) ⇒

(define (f_{42} m) ( + 3 m)) (f_{42} 4) ⇒

(+ 3 4) ⇒ 7
A note on scope:

\[
\begin{align*}
\text{(define (add3 m) &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp; (define (make-adder n) &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&amp...}
\end{align*}
\]

In \textit{add3} the parameter \textit{m} is of no consequence after \textit{add3} is applied. Once \textit{add3} produces its value, \textit{m} can be safely forgotten.

However, our earlier trace of \textit{make-adder} shows that after it is applied the parameter \textit{n} does have a consequence. It is embedded into the result, \textit{f}, where it is “remembered” and used again, potentially many times.
Binding functions to identifiers

The result of `make-adder can be bound to an identifier` and then used repeatedly.

```scheme
(define add2 (make-adder 2))
(define add3 (make-adder 3))

(add2 3) ⇒ 5
(add3 10) ⇒ 13
(add3 13) ⇒ 16
```
How does this work?

(define add2 (make-adder 2)) ⇒
(define add2 (local [(define (f m) (+ 2 m))] f)) ⇒
(define (f_43 m) (+ 2 m)) ; rename and lift out f
(define add2 f_43)

(add2 3) ⇒
(f_43 3) ⇒
(+ 2 3) ⇒
5
Putting functions in lists

Recall our code in lecture module 08 for evaluating alternate arithmetic expressions such as '\((+ (* 3 4) 2)\).

;; eval: AltAEExp → Num
(define (eval aax)
  (cond [(number? aax) aax]
        [else (my-apply (first aax) (rest aax))])))
;; my-apply: Sym AltAExpList → 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 06 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) ⇒ 12
;; newapply: Sym AltAExpList → 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.

(+) ⇒ 0 and (∗) ⇒ 1
;; newapply: Sym AltAExpList → Num
(define (newapply f aaxl)
  (local [(define op (lookup-al f trans-table))]
    (cond [(empty? aaxl) (op)]
      [else (op (eval (first aaxl))
        (newapply f (rest aaxl)))])))

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.
Key Point: We can use the contract for a function as its type.

For example, the type of \( > \) is \((\text{Num Num} \to \text{Bool})\), 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.
An example:

\[
\text{(define trans-table (list (list ' + + )} \\
\qquad \quad (\text{list ' \ast \ast }))\]

;; (lookup-al k alst) finds the value in alst corresponding to key k
;; lookup-al: Sym (listof (list Sym (Num Num → Num))) → \\
;; (anyof false (Num Num → Num))
(define (lookup-al k alst)
  (cond [(empty? alst) false]
        [(equal? k (first (first alst))) (second (first alst))]
        [else (lookup-al k (rest alst))])))
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

(Any → Bool) (listof Any) → (listof Any).

But this is not specific enough.

Consider the application (filter odd? (list 1 2 3)). This does not obey the contract (the contract for odd? is Int → Bool) but still works as desired.

The problem: there is a relationship between 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.

*Key Point:* 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 dependency, 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 \((X \to \text{Bool})\), 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:

```;
;; filter: (X \to \text{Bool}) (\text{listof } X) \to (\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. Here the type variable is used to indicate a relationship. In other cases (e.g. Module 6) when we talk about a list, we might use the term (listof X) to mean the elements of the list are all the same type. We will soon see examples where more than one type variable is needed in a contract.
Using contracts to understand

*Key Point:* 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.
Simulating structures *Not covered this term!*

We can use the ideas of producing and binding functions to simulate structures.

```scheme
(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.
```
Simulating structures *Not covered this term!*

```
(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.
Simulating structures *Not covered this term!*

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 'x) ⇒ 3
```
Simulating structures *Not covered this term!*

We can define a few convenience functions to simulate `posn-x` and `posn-y`:

```lisp
(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.
Simulating structures *Not covered this term!*

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

(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))

The result of evaluating this expression is a function.

What is its name? It is anonymous (has no name).

When you do calculations such as (* (+ 2 3) 4) the intermediate result 5 also does not have a name.

This is sufficiently valuable that there is a special mechanism for it.
Producing anonymous functions

(\text{define} \ (\text{not-symbol-apple?} \ \text{item}) \ (\text{not} \ (\text{symbol}=? \ \text{item} \ \text{'apple})))

(\text{define} \ (\text{eat-apples} \ \text{lst}) \ (\text{filter} \ \text{not-symbol-apple?} \ \text{lst}))

This is a little unsatisfying, because \text{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 \text{local} expressions.
Using local

(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.
But this is still unsatisfying. *Why should we have to name not-symbol-apple?* at all? In the expression \((\ast (\plus 2 3) 4)\), 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 is used to create anonymous functions.

lambda can be thought of as “make-function”.

Key Point: 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

\[
\text{(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
      (not (symbol = ? item 'apple)))])
    not-symbol-apple?))
  lst)
\]

with the following:

\[
\text{(define (eat-apples lst)
  (filter (lambda (item) (not (symbol = ? item 'apple))) lst))}
\]
**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 \texttt{lambda} to simplify \texttt{make-adder}. Instead of

\begin{verbatim}
(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))
\end{verbatim}

we can write:

\begin{verbatim}
(define (make-adder n)
  (lambda (m) (+ n m)))
\end{verbatim}

\textit{Key Point:} \texttt{lambda} replaces \texttt{local}, \texttt{define} and the identifier \texttt{foo} of an anonymous function but keeps the argument(s) and body.
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, where the value may be a function.
Internally,

\[(\text{define} \ (\text{interest-earned} \ \text{amount}) \n\quad (\ast \ \text{interest-rate} \ \text{amount}))\]

is translated to

\[(\text{define} \ \text{interest-earned} \n\quad (\text{lambda} \ (\text{amount}) \ (\ast \ \text{interest-rate} \ \text{amount})))\]

which binds the name \text{interest-earned} to the value

\[(\text{lambda} \ (\text{amount}) \ (\ast \ \text{interest-rate} \ \text{amount})).\]
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.

In stepper questions, if a function is defined using function syntax, you can skip the lambda substitution step. If a function is defined as a constant using lambda, you must include the lambda step.
For example, here’s `make-adder` rewritten using `lambda`.

```
(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

What is `((make-adder 3) 4)`?
\begin{quote}
(define make-adder (lambda (x) (lambda (y) (+ x y))))

((make-adder 3) 4) \Rightarrow ;; substitute the lambda expression

(((lambda (x) (lambda (y) (+ x y))) 3) 4) \Rightarrow

((lambda (y) (+ 3 y)) 4) \Rightarrow

(+ 3 4) \Rightarrow 7

\end{quote}

*make-adder* is defined as a constant using lambda, so it is substituted in place of *make-adder.*
Syntax and semantics of lambda

Before

*First position* in an application must be a built-in or user-defined function.

A function name had to follow *an open parenthesis*.

Now

*First position* can be an expression (computing the function to be applied). Evaluate it along with the other arguments.

A function application can have *two or more open parentheses* in a row: ((make-adder 3) 4).
Semantics of lambda

Key Point: We need a rule for evaluating applications where the function being applied is anonymous (a lambda expression.) The rule for evaluating

\[ ((\text{lambda} \ (x_1 \ldots \ x_n) \ \text{exp}) \ v_1 \ldots \ v_n) \Rightarrow \text{exp}' \]

is that exp’ is exp with all occurrences of x1 replaced by v1, all occurrences of x2 replaced by v2, and so on.

As an example:

\[ ((\text{lambda} \ (x \ y) \ (\ast \ (\div \ y \ 4) \ x)) \ 5 \ 6) \Rightarrow (\ast \ (\div \ 6 \ 4) \ 5) \]
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

We’d like a function, translate, that translates one string into another according to a set of rules that are specified when it is applied.

In one application, we might want to change every instance of ’a’ to a ’b’. In another, we might translate lowercase characters to the equivalent uppercase character and digits to ”*”.

(check-expect (translate "abracadabra" ...) "bbrbcdbbbrb")
(check-expect (translate "Testing 1-2-3" ...) "TESTING *-*-*")

We use . . . to indicate that we still need to supply some arguments.
We could imagine `translate` containing a `cond`:

```
(cond [(char = ? ch #\a) #\b]
      [(char-lower-case? ch) (char-upcase ch)]
      [(char-numeric? ch) #\*]
      ...)
```

But this fails for a number of reasons:

- The rules are “hard-coded”; we want to supply them when `translate` is applied.

- A lower case ’a’ would always be translated to ’b’; never to ’B’

But the idea is inspiring...
Goal: develop a general method of performing character translations on strings.

Suppose we supplied translate with a list of question/answer pairs:

;; A TranslateSpec is one of:
;; * empty
;; * (cons (list Question Answer) TranslateSpec)

Like cond, we could work our way through the TranslateSpec with each character. If the Question produces true, then apply the Answer to the character. If the Question produces false, go on to the next Question/Answer pair.

What are the types for Question and Answer?
Functions as first class values can help us. Both Question and Answer are functions that consume a Char.

Question produces a Bool and Answer produces a character. This completes our data definition, above:

;; A Question is a Char → Bool
;; An Answer is a Char → Char

And a completed example:

(check-expect (translate "Testing 1-2-3"
                 (list (list char-lower-case? char-upcase)
                       (list char-numeric? (lambda (ch) #\*))))
             "TESTING *--*--*")
Translate: developing the code

translate consumes a string and produces a string but we need to operate on characters. This suggests a wrapper function:

;; A TranslateSpec is one of:
;; * empty
;; * (cons (list Question Answer) TranslateSpec)

;; (translate s spec) translates s according to the spec
;; translate: Str TranslateSpec → Str
(define (translate s spec)
  (list→string (trans-loc (string→list s) spec)))
;; trans-loc (listof Char) TranslateSpec → (listof Char)
(check-expect (trans-loc (list #\a #\9)
  (list (list char-lower-case? char-upcase))) (list #\A #\9))

(define (trans-loc loc spec)
  (cond [(empty? loc) empty]
        [(cons? loc) (cons (trans-char (first loc) spec)
                             (trans-loc (rest loc) spec))]))

(define (trans-char ch spec)
  (cond [(empty? spec) ch]
        [((first (first spec)) ch) ((second (first spec)) ch)]
        [else (trans-char ch (rest spec))])))
(check-expect (translate "Testing 1-2-3"
    (list (list char-lower-case? char-upcase)
          (list char-numeric? (lambda (ch) #\*)))) "TESTING *-***")

(check-expect (translate "abracadabra"
    (list (list (lambda (ch) (char= ch #\a))
              (lambda (ch) #\b)))) "bbrbcdbbbrb")

The repeated lambda expressions suggest some utility functions:

(define (is-char? c1) (lambda (c2) (char= c1 c2)))
(define (always c1) (lambda (c2) c1))
Abstracting another set of 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 payroll)
  (cond [(empty? payroll) empty]
    [else (cons (sr→tr (first payroll))
                     (compute-taxes (rest payroll)))]))
The function **map**

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.

I.e. in both cases we are applying a function \( f \) to each element of the list, so consume \( f \) as a parameter.

;; (my-map f lst) applies f to evey element of lst

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

CS 135 Fall 2019 13: Functional abstraction 63
Tracing my-map

(my-map sqr (list 3 6 5))
⇒ (cons 9 (my-map sqr (list 6 5)))
⇒ (cons 9 (cons 36 (my-map sqr (list 5)))))
⇒ (cons 9 (cons 36 (cons 25 (my-map sqr empty)))))
⇒ (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.
Using **my-map**

The application

\[(\text{my-map } f \ (\text{list } x_1 \ x_2 \ldots \ x_n))\]

has the same effect as evaluating

\[(\text{list } (f \ x_1) (f \ x_2) \ldots (f \ x_n)).\]

We can use **my-map** to give short definitions of a number of functions we have written to consume lists:

\[
\begin{align*}
\text{(define (negate-list lst) (my-map } - \text{ lst))} \\
\text{(define (compute-taxes lst) (my-map sr→tr lst))}
\end{align*}
\]

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. Check out the Help Desk (in DrRacket, Help → Help Desk → How to Design Programs Languages → 4.17 Higher-Order Functions)

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.
Abstracting another set of examples

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)))]))
Abstracting another set of examples

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 (list-template lst)
  (cond [(empty? lst) . . .]
    [else (. . . (first lst) .
        (list-template (rest lst)) . . .)])
)

We replace the first ellipsis by a \textit{base value}.

We replace the rest of the ellipses by some \textit{function which combines} \texttt{(first lst)} and the result of a recursive function application on \texttt{(rest lst)}.

\textit{Key Point:} This suggests passing the \textit{base value} and the \textit{combining function} as parameters to an abstract list function.
The abstract list function \texttt{foldr}

\begin{verbatim}
(define (my-foldr combine base lst)
  (cond 
    [(empty? lst) base]
    [else (combine (first lst)
                     (my-foldr combine base (rest lst)))]))
\end{verbatim}

\texttt{foldr} is also a built-in function in Intermediate Student With Lambda.
Tracing \textit{my-foldr}

\[(\text{my-foldr } f \ 0 \ (\text{list } 3 \ 6 \ 5)) \Rightarrow \]
\[(f \ 3 \ (\text{my-foldr } f \ 0 \ (\text{list } 6 \ 5))) \Rightarrow \]
\[(f \ 3 \ (f \ 6 \ (\text{my-foldr } f \ 0 \ (\text{list } 5)))) \Rightarrow \]
\[(f \ 3 \ (f \ 6 \ (f \ 5 \ (\text{my-foldr } f \ 0 \ \text{empty})))) \Rightarrow \]
\[(f \ 3 \ (f \ 6 \ (f \ 5 \ 0))) \Rightarrow \ldots \]

Intuitively, \textit{the effect} of the application

\[(\text{foldr } f \ b \ (\text{list } x_1 \ x_2 \ldots \ x_n)) \] is to compute the value of the expression

\[(f \ x_1 \ (f \ x_2 \ (\ldots \ (f \ x_n \ b) \ldots ))).\]
More on `my-foldr`

`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:

1. a function which combines the first list item with the result of reducing the rest of the list;

2. a base value;

3. a list on which to operate.

What is the contract for `foldr`?

```
;; foldr: (X Y → Y) Y (listof X) → Y
```
Using \texttt{foldr} to sum a list

\begin{verbatim}
(define (sum-of-numbers lst) (foldr + 0 lst))
\end{verbatim}

If \texttt{lst} is (\texttt{list x1 x2 \ldots xn}), then by our intuitive explanation of \texttt{foldr}, the expression \texttt{(foldr + 0 lst)} reduces to

\begin{verbatim}
(+ x1 (+ x2 (+ \ldots (+ xn 0) \ldots)))
\end{verbatim}

Thus \texttt{foldr} \textit{does all the work of the template for processing lists}, in the case of \texttt{sum-of-numbers}.
Using \texttt{foldr}

The function provided to \texttt{foldr} consumes two parameters:

1. one is an element on the list which is an argument to \texttt{foldr}, and

2. 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 \texttt{foldr} to compute \texttt{count-symbols}.
Using \texttt{foldr} to count symbols

The important thing about the \textit{first argument} to the function provided to \texttt{foldr} is that it \textit{contributes 1 to the count}; its actual value is irrelevant.

Thus the function provided to \texttt{foldr} in this case can ignore the value of the first parameter, and \textit{just add 1 to the reduction of the rest of the list}.
Using **foldr** to count symbols

\[
\text{(define (count-symbols lst) (foldr (lambda (x rror) (add1 rror)) 0 lst))}
\]

The function provided to **foldr**, namely

\[
\text{(lambda (x rror) (add1 rror))}
\]

*ignores its first argument, x.*

Its second argument is the result of recursing on the rest (**rror**) of the list (in this case the length of the rest of the list, to which 1 must be added).
More examples

What do these functions do?

\[
\text{(define (bar lon)} \\
\quad \text{(foldr max (first lon) (rest lon)))}
\]

\[
\text{(bar } '(1 5 23 3 99 2))
\]

\[
\text{(define (foo los)} \\
\quad \text{(foldr (lambda (s rror) (+ (string-length s) rror)) 0 los))}
\]

\[
\text{(foo } '("one" "two" "three"))
\]
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.

**Key Idea:** define a function \((\text{lambda} \ (x \ rror) \ldots)\) where \(x\) is the first element of the list and \(rror\) 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.
Using \texttt{foldr} to implement \texttt{map}

The function we need is

\[\text{(}\lambda (x \operatorname{rror}) \operatorname{cons} (- x) \operatorname{rror}\text{)}\]

Thus we can give a nonrecursive version of \texttt{negate-list} (that is, \texttt{foldr} does all the recursion).

\[\text{(define (negate-list lst)}}\]
\[\text{(foldr (lambda (x rror) (cons (- x) rror)) empty lst))}\]

\textit{Key Observation:} Because we generalized \texttt{negate-list} to \texttt{map}, we should be able to use \texttt{foldr} to define \texttt{map}.
Recall: the implementation of \textit{map}

Let's look at the code for \textit{my-map}.

\begin{verbatim}
(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                   (my-map f (rest lst)))]))
\end{verbatim}

Clearly \textit{empty} is the base value, and the function provided to \textit{foldr} is something involving \textit{cons} and \textit{f}.
Recall: the implementation of map using foldr

In particular, the function provided to foldr must apply \( f \) to its first argument, then \textbf{cons} the result onto its second argument (the reduced rest of the list).

\[
\text{(define (my-map } f \text{ lst)}
\]
\[
\quad \text{(foldr (lambda (x rror) (cons (f x) rror)) empty lst))}
\]

We can also implement \texttt{my-filter} using foldr.
Abstract list functions

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

Key Point: 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.
Abstract list functions

*Key Point:* 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.

Shorter code is not always better!
Generalizing accumulative recursion: ex 1

This function (from Mod 12-31) uses recursion (with an accumulator) on a list.

;; code from module 12

(define (sum-list lon)
  (local [(define (sum-list/acc lst sum-so-far)
            (cond [(empty? lst) sum-so-far]
                  [else (sum-list/acc (rest lst) (+ (first lst) sum-so-far))]))]
    (sum-list/acc lon 0)))
Generalizing accumulative recursion: ex 2

This function (from Mod 13-88) uses recursion (with an accumulator) on a list.

;; code from lecture module 9-14 rewritten to use local

(define (my-reverse lst0)
  (local [(define (my-rev/acc lst list-so-far)
    (cond [(empty? lst) list-so-far]
      [else (my-rev/acc (rest lst) (cons (first lst) list-so-far))])]
    (my-rev/acc lst0 empty)))
)
Contrasting: ex 1 and ex 2

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*. 
Introducing foldl

\[
\begin{align*}
\text{(define (my-foldl combine base lst0)} & \text{ (local [(define (foldl/acc lst acc)
}\n\text{ (cond [(empty? lst) acc]
}\text{ [else (foldl/acc (rest lst)
}\text{ (combine (first lst) acc))]])]
\text{(foldl/acc lst0 base)))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (sum-list lon) (my-foldl + 0 lon))}
\text{(define (my-reverse lst) (my-foldl cons empty lst))}
\end{align*}
\]
We noted earlier that intuitively, the effect of the application

\((\text{foldr } f \ b \ (\text{list } x_1 \ x_2 \ldots \ x_n))\)

is to compute the value of the expression

\((f \ x_1 \ (f \ x_2 \ (. \ldots \ (f \ x_n \ b) \ldots))))\)

What is the intuitive effect of the following application of \text{foldl}?

\((\text{foldl } f \ b \ (\text{list } x_1 \ldots \ x_{n-1} \ x_n))\)

The function \text{foldl} is provided in Intermediate Student.

What is the contract of \text{foldl}?
foldr vs. foldl

- \((\text{foldr } f \ b \ \text{(list } x_1 \ x_2 \ \ldots \ x_n))\) computes \((f \ x_1 \ (f \ x_2 \ (\ldots \ (f \ x_n \ b)\ldots)))\).

- **foldr** starts calculating from the right side of the list.

- \((\text{foldl } f \ b \ \text{(list } x_1 \ x_2 \ \ldots \ x_n))\) computes \((f \ x_n \ (f \ x_{n-1} \ (\ldots \ (f \ x_1 \ b)\ldots)))\).

- **foldl** starts calculating from the left side of the list.

- The contract is the same for both \((X \ Y \rightarrow Y) \ Y \ (\text{listof } X) \rightarrow Y\).
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

\[
\text{list (f 0) (f 1) \ldots (f (sub1 n))}
\]

\[
\text{(build-list 4 (lambda (x) x)) \Rightarrow (list 0 1 2 3)}.
\]

Clearly `build-list` abstracts the “count up” pattern, and it is easy to write our own version.
An implementation of \texttt{build-list}

\begin{verbatim}
(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)))
\end{verbatim}
Build-list example: \texttt{sum}

\[
\sum_{i=0}^{n-1} f(i):
\]

\begin{verbatim}
(define (sum n f)
  (foldr + 0 (build-list n f)))
\end{verbatim}

\begin{verbatim}
(sum 4 sqr) \Rightarrow 14
\end{verbatim}
Build-list example: **mult-table**

We can now simplify **mult-table** even further.

```
(define (mult-table nr nc)
  (build-list nr
    (lambda (r)
      (build-list nc
        (lambda (c)
          (∗ r c))))))
```
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 \texttt{lambda}, bound to identifiers, and placed in lists.

You should be familiar with the following built-in abstract list functions provided by Racket (\texttt{filter}, \texttt{(map}, \texttt{(foldl}, \texttt{(foldr}, \texttt{(build-list)} 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.
Module 13 Summary

Functions a First-class Values in Racket

1. **Abstraction** is the process of finding similarities or common aspects, and forgetting unimportant differences. [2]

2. In particular consider the general structure of two similar functions. [5]

3. In Racket functions are first-class values. [6]

4. **First-class** values are values that can be consumed as arguments [8-11], produced as results [13-16], bound to identifiers [18-19], and put in lists and structures [20-25].
Module 13 Summary

Functional Abstraction and their Contracts

5. **Functional abstraction** is the process of combining related functions into a single definition (such as filter). [12]

6. The body of `local` can be used to create functions [13-16] and `define` can be used to bind the results to an identifier. [18-19]

7. Use the contract for a function as its type. [26-27]

8. To capture the dependency between the type of the predicate and the type of list, use a type variable such as `X`. [30]

9. A function is **polymorphic** or **generic** if it works on many types of data. [31-33]
Module 13 Summary

Lambda

10. We skipped slides 34-38.

11. **Anonymous functions** are functions that do not have a name. [39]

12. Use `lambda` to produce anonymous functions (which can then be used as an argument). [43-51]

13. `lambda` replaces `local`, `define` and the identifier of an anonymous function but keeps the arguments and body. [46]

14. `define` binds a name to a value (where the value may be a function or a constant). [47-48]
Module 13 Summary

Syntax and Semantics of Lambda

15. During a trace, the value of a function does not get rewritten as a lambda expression. [49]

16. The first position in an application can be an expression computing a function, e.g. \((\text{make-adder } 3) 4\), or a named function, e.g. \((\text{sum1 } 4)\). [52]

17. The rule for evaluating \(((\text{lambda}(x_1 \ldots x_n) \text{ exp}) \; v_1 \ldots v_n) \Rightarrow \text{ exp'}\) is that 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. [53]

18. Note that exp can be another lambda expression. [50]
Module 13 Summary

Abstract List Functions filter, map and foldr

19. filter is a built-in function that consumes a predicate \((X \rightarrow \text{Bool})\) and a \((\text{listof } X)\) and produces a \((\text{listof } X)\) by selecting elements in the list where the predicate produces true. [60]

20. The built-in abstract list function \((\text{map } f \ \text{lst})\) applies \(f\) to each element of \(\text{lst}\) to produce a new list. [62-65]

21. Its contract is \(\text{map}: (X \rightarrow Y) \ (\text{listof } X) \rightarrow (\text{listof } Y)\) [66]

22. The built-in abstract list function \((\text{foldr } \text{combine } \text{base } \text{lst})\) has a base value, \(\text{base}\), a combining function, \(\text{combine}\) and a \(\text{lst}\) to create a new value. [68-80]
Module 13 Summary

Abstract List Functions: foldr and foldl

23. Its contract is `foldr: (X Y → Y) Y (listof X) → Y` [75]

24. `foldr` can be used with `cons` to produce lists. [81-84]

25. The higher-order function `(foldl combine base list)` generalizes simple recursion on a list with one accumulator. [90]

26. `foldl` consumes

   (a) `combine`: a function that computes of the new value of the accumulator,

   (b) `base`: the initial value of the accumulator,

   (c) `list`: a list. [90]
Module 13 Summary

Abstract List Functions: foldl and build-list

27. Its contract is \( \text{foldl}: (X \rightarrow Y) \rightarrow Y \rightarrow Y \rightarrow Y \) \([91.1]\)

28. Functions that consume or produce functions are called **higher-order functions**. \([92]\)

29. \text{build-list} is a built-in higher-order function that takes a function \( f \) and a natural number \( n \) and produces the list \( \text{list} (f 0) (f 1) (f 2) \ldots (f (\text{sub1} n)) \). \([93]\)
Generative recursion

Readings: Sections 25, 26, 27, 30, 31

Topics:

• What is generative recursion? [2–4]
• Termination [5–10]
• Hoare’s Quicksort [11-17]
• Modifying the design recipe [18]
• Example: breaking strings into lines [19-28]
What is generative recursion?

Simple and accumulative 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 → Nat

(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))

E.g.

gcd(100, 85) ⇒ gcd(85, 15) ⇒ gcd(15, 10) ⇒ gcd(10, 5)
⇒ gcd(5, 0) ⇒ 5
Why does this work?

**Correctness:** Follows from Math 135 proof of the identity.

**Key Idea:** **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 Racket 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 functions using simple recursion terminate?

*Key Observation:* a simple 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.
Example: termination for simple recursion

$$(\text{sum-list} \ (\text{list} \ 3 \ 6 \ 5 \ 4)) \Rightarrow$$
$$\ (\ + \ 3 \ (\text{sum-list} \ (\text{list} \ 6 \ 5 \ 4))) \Rightarrow$$
$$\ (\ + \ 3 \ (\ + \ 6 \ (\text{sum-list} \ (\text{list} \ 5 \ 4)))) \Rightarrow \ldots$$

The depth of recursion of any application of $\text{sum-list}$ is equal to the length of the list to which it is applied.

*Key Point:* For generatively recursive functions, we need to make a similar argument.
Example: termination of euclid-gcd

The function euclid-gcd terminates for any natural number.

*Key Idea:* 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, which makes the second argument smaller.

After that, *the second argument* is always smaller than the first argument in any recursive application, due to the application of the remainder modulo $m$. 

Example: termination of euclid-gcd

The second argument always gets smaller in the recursive application (since $m > n \mod m$), 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 \geq 1
(define (collatz n)
  (cond [(= n 1) 1]
        [(even? n) (collatz (/ n 2))]
        [else (collatz (+ 1 (* 3 n)))]))

Key Point: It is a decades-old open research problem (i.e. we currently don’t know the answer) 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 → (listof Nat)
;; requires: n ≥ 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)))])))
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 or equal to* the pivot (or just greater than the pivot if duplicated values are not allowed).
Quicksort: example

If the list is \((\text{list } 9 \ 4 \ 15 \ 2 \ 12 \ 20)\), and the pivot is 9, then the subproblems are \((\text{list } 4 \ 2)\) and \((\text{list } 15 \ 12 \ 20)\).

Recursively sorting the two subproblem lists gives \((\text{list } 2 \ 4)\) and \((\text{list } 12 \ 15 \ 20)\).

It is now simple to combine them with the pivot to give the answer.

\((\text{append } (\text{list } 2 \ 4) \ (\text{list } 9) \ (\text{list } 12 \ 15 \ 20))\)
Quicksort: selecting the pivot

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 or equal to the pivot).
quick-sort implementation

;; (quick-sort lon) sorts lon in non-decreasing order
;; quick-sort: (listof Num) → (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)))]))
Quicksort: termination

Quicksort terminates because 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

\[(\text{filter} \ (\lambda (x) \ (> \!\!= x \ \text{pivot})) \ \text{lon})\]

instead of the correct

\[(\text{filter} \ (\lambda (x) \ (> \!\!= x \ \text{pivot})) \ \text{(rest lon)})\]
Quicksort in Racket

In the teaching languages, the built-in function quicksort (note no hyphen) consumes two arguments, a list and a comparison function.

(quicksort '(1 5 2 4 3) <) ⇒ '(1 2 3 4 5)
(quicksort '(1 5 2 4 3) >) ⇒ '(5 4 3 2 1)
Quicksort efficiency

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.
Modifying the design recipe

The design recipe becomes much more vague when we move away from data-directed design.

*Note:* The purpose statement remains unchanged, but additional documentation is often required to describe *how* the function works.

Examples need to illustrate the workings of the algorithm.

We cannot apply a template, since there is no data definition.

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: new lines

The character sets used in computers include “control” characters.

In Racket the character \newline, signals the start of a new line of text.

In a string ‘\’ and ‘n’ appearing consecutively in a string constant are interpreted as a single newline character (i.e. as \newline).

For example, the string "ab\ncd" is a five-character string with a newline as the third character. It would typically be printed as "ab" on one line and "cd" on the next line.
New line: need for generative recursion

Consider converting a string such as "one\ntwo\nthree" into a list of strings, \((\text{list } "\text{one}" "\text{two}" "\text{three}"),\) one for each line.

The solution will start with an application of \texttt{string→list}. That’s the only way we’ve studied of working with individual characters in a string.

This problem can be solved using simple recursion on the resulting list of characters – but it’s hard. The “simple” recursion \textit{gets bogged down in a lot of little details}.

In this case a generative solution is easier.
The generative idea

*Approach:* Instead of thinking of the list of characters as a list of characters, think of it as a list of lines:

```
one
  two
  three
none
two
three
```

The *string version* is "\n" and the *char version* is #\newline.

A list of lines is either empty or a line followed by a list of lines.

So start with helper functions first-line that divide the list of characters into the first line and the rest of the lines.
Example: breaking strings into lines

- **As a string** the newline is written as “\n”, e.g. “ab\ncd”.

- **As a list of characters** the newline is written as #\newline, e.g. (list #\a #\b #\newline #\c #\d).

- Slide 23: The definition of **first-line**, i.e. produce the **first line** from the list of characters, **loc**.

- Slide 24: The definition of **rest-of-lines**, i.e. produce the **rest of the lines** from the list of characters, **loc**.

- Slide 25: The template that uses **first-line** and **rest-of-lines**.

- Slide 26: The function that uses **first-line** and **rest-of-lines** to break up the list of characters into a list of lines.

- Slide 29: a simple recursion version.
;; (first-line loc) produces longest newline-free prefix of loc
;; first-line: (listof Char) → (listof Char)

;; Examples:
(check-expect (first-line empty) empty)
(check-expect (first-line ’(#\a #\newline)) ’(#\a))
(check-expect (first-line (string→list "abc\ndef")) ’(#\a #\b #\c))

(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) → (listof Char)

;; Examples:
(check-expect (rest-of-lines empty) 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))])))
List of lines template

We can create a “list of lines template” using these helpers.

\[
\text{(define (loc→lol loc))}
\]

\[
\text{(local)
\[(\text{define fline (first-line loc))}
\text{)(define rlines (rest-of-lines loc))\]}
\text{(cond [(empty? loc) empty]
\[\text{else \ldots fline \ldots (loc→lol rlines)} \ldots ])}
\]
;; list→lines: (listof Char) → (listof Str)
(check-expect (loc→lol (string→list "abc
ndef")) (list "abc" "def"))
(check-expect (loc→lol (string→list " ")) (list))
(check-expect (loc→lol (string→list "
ndef")) (list " " "def"))

(define (loc→lol loc)
  (local [(define fline (first-line loc))
           (define rlines (rest-of-lines loc))]
    (cond [(empty? loc) empty]
          [else (cons (list→string fline)
                      (loc→lol rlines))])))
Generative recursion

Why is this generative recursion?

loc → lol can be rewritten as

\[
\text{(define (loc → lol loc)}
\]
\[
\text{ (cond [(empty? loc) empty]}
\]
\[
\text{ [else (cons (list → string (first-line loc))}
\]
\[
\text{ (loc → lol (rest-of-lines loc)))])}
\]

The recursive call to \text{loc → lol} is not using the data definition for a list of characters. It often gets many steps closer to the base case in one recursive application.
It *is* using a data definition of a “list of lines”, but that’s a higher-level abstraction that we imposed on top of the (listof Char), our actual argument.

The key part of the generative recursion pattern is that the argument to \texttt{loc→lol} is being generated by \texttt{rest-of-lines}.

With generative recursion we needed that “aha” that transformed the problem into a list of lines. That “aha” is often difficult to see.

Was it worth it? Consider the solution using “simple” recursion on the next slide. This still needs a wrapper function to do both pre- and post-processing.
Simple recursion version *Not covered this term!*

```
(define (list→lines loc)
  (cond [(empty? loc) (list empty)] ; 2
    [(and (empty? (rest loc)) (char=\? #\newline (first loc))) ; 3
      (list empty)] ; 4
    [else
      (local [(define r (list→lines (rest loc)))] ; 6
        (cond
          [(char=\? (first loc) #\newline) (cons empty r)] ; 8
          [else (cons (cons (first loc) (first r)) ; 9
                         (rest r))]))])) ;10
```

CS 135 Fall 2019 14: Generative recursion 28
Simple recursion version *Not covered this term!*

**Key Idea:** Each time you see a newline start a new list to add the characters to.

E.g. for \( \text{(list #\ a #\ b #\ \text{newline} #\ c #\ d)} \)

- Base case: start with \( \text{(list empty)} \).
- Cons d then c to the inner list, i.e. \( \text{(list (list c d)} \)
- For newline, cons a new empty list c to the inner list, i.e. \( \text{(list empty (list c d)} \)
- Cons b then a to the first inner list, i.e. \( \text{(list (list a b) (list c d)} \)
Simple recursion version  *Not covered this term!*

- Line 2: Base case: create a list of lines with one empty line in it.
- Lines 3-4: If the last character is a newline, do the same as above.
- Line 6: define $r$ locally to avoid repeated recursive applications
- Line 8: if the char is a newline then cons a new line (i.e. a new list) to the list of lines.
- Lines 9-10: otherwise add the char, i.e. (first loc), to the first list in the list of lines.
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.
Module 14 Summary

Simple and Generative Recursion

1. Simple recursion is a way of deriving code whose form parallels a data definition. [2]

2. **Generative recursion** is more general: the recursive cases are generated as a function of the arguments. [2]

3. An application **terminates** if it can be reduced to a value in finite time. [4]

4. A simple recursive function always makes recursive applications on smaller instances, whose size is bounded below by the base case. [5]
Module 14 Summary

Generative Recursion

5. The **depth of recursion** is the number of recursive applications of the function until the base case is reached. [5]

6. For generative recursion, an argument has to be made that the recursion will terminate. [6]

7. Calculating the **euclidean-gcd** of two numbers is an example of generative recursion. [7–8]

8. It is not currently known if the **collatz** function terminates for all non-zero natural numbers. [9–10]
Module 14 Summary

Quicksort

9. A **divide and conquer** algorithm is an algorithm that
   (a) divides a problem into smaller subproblems;
   (b) recursively solves each one;
   (c) combines the solutions to solve the original problem. [11]

10. **quicksort** is a divide and conquer algorithm.

11. The subproblems are the elements less than the pivot and those greater than or equal to the pivot. [11]

12. Termination occurs because both subproblems are smaller than the original list. [15]
Module 14 Summary

Generative Recursion and Breaking up a String

13. For generative recursion, design recipe often requires more documentation about how the function works. [18]

14. Generative recursion can be used to break a string up into a list of lines, i.e. $\text{loc} \rightarrow \text{lol}$. [19–27]
Graphs

Readings: Section 28

Topics:

• Directed graphs [2–5]
• Representing graphs [6–9]
• Backtracking and find-route [10–26]
• Termination [27-29]
• Avoiding cycles: find-route v2.0 [30–39]
• Improving efficiency: find-route v3.0 [40-46]
Directed graphs

A directed graph consists of a collection of vertices (also called nodes) together with a collection of edges.

An edge is an ordered pair of vertices, which we can represent by an arrow from one vertex to another.
Example of a directed graph

There are 9 nodes: A, B, C, D, E, F, H, J, K

There are 11 edges: (A,C), (A,D), (A,E), (B,E), (B,J), (D,F), (D,J), (E,K), (F,K), (F,H), (J,H)
Other Examples

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 vertices \(v_1, v_2, \ldots, v_k\) is a path or route of length \(k - 1\) if \((v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)\) are all edges.

If \(v_1 = v_k\) (i.e. the path starts and ends on the same node, this is called a cycle).

Directed graphs without cycles are called DAGs (directed acyclic graphs).
Representing graphs

*Key Idea:* 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.

*Key Idea:* 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 an adjacency list

\[((A (C D E))
(B (E J))
(C ())
(D (F J))
(E (K))
(F (K H))
(H ())
(J (H))
(K ()))\]
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)))
The template for graphs

;; graph-template: Graph → Any
(define (graph-template G)
  (cond
    [(empty? G) . . .]
    [(cons? G)
      ( . . . (first (first G)) . . . ; first node in graph list
       ;; list of adjacent nodes
       (listof-node-template (second (first G))) . . .
       (graph-template (rest G)) . . .)]))
Finding routes

A path can be represented by the list of nodes on the path.

**Goal:** Design a function `find-route` that 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, i.e. `(find-route orig dest G)`.

**First step:** we create an auxiliary function `neighbours` that consumes a node and a graph and produces the list of out-neighbours of the node, i.e. `(neighbours v G)`.
Neighbours in our example

(neighbours 'A G) ⇒ (list ’C ’D ’E)
(neighbours ’D G) ⇒ (list ’F ’J)
(neighbours ’H G) ⇒ empty
Helper function: neighbours

;; (neighbours v G) produces list of neighbours of v in G
;; neighbours: Node Graph → (listof Node)
;; requires: v is a node in G
(define (neighbours v G)
  (cond [(symbol=? v (first (first G))) (second (first G))]
        [else (neighbours v (rest G))])))

Algorithm: recursively search for v in the adjacency list (the first element of the pair) and produce the list of v’s out-neighbours, i.e. the second element in the (listof Node (listof Node)) pair.
Cases for **find-route**

Structural recursion does not work for **find-route**; we must use generative recursion.

1. **Base Case:** If the origin equals the destination, the path consists of just this node.

2. **Recursive Case:** Otherwise, if there is a path, the second node on that path must be an out-neighbour of the origin node, so search there.

**Key Idea:** Each out-neighbour defines a subproblem (finding a route from it to the destination).
Examples of subproblems

In our example, any route from A to H must pass through C, D, or E (i.e. the *out-neighbours of A*).

*Subproblems:* If we knew a route (1) from C to H, or (2) from D to H, or (3) from E to H, we could create one from A to H.
Backtracking algorithms

Backtracking algorithms try to find a route from an origin to a destination.

Key Idea: If the initial attempt (subproblem) does not work, the algorithm backtracks and tries another choice.

Termination: Eventually, either a route is found, or all possibilities are exhausted, meaning there is no route.
Backtracking in out example

In our example, we can see the backtracking since the search for a route from A to H can be seen as moving forward in the graph looking for H.

*Key Point:* Starting at A, if the search fails (for example, at C), then the algorithm *backs up to the previous vertex*, A, and tries A’s next neighbour, D.

*If this choice succeeds* i.e. it finds a path from D to H, add A to the beginning of this path.
Second step: creating `find-route/list`

*Key Tasks:* 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 node’s out-neighbour on that list until it finds a route to the destination.
Parallels with expression trees

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`. 
Overview of **find-route** and **find-route/list** v1.0

**find-route**: take step

- **take one step** forward (i.e. create list of out-neighbours and check them with **find-route/list**)
- if step was successful, then add **orig** to path, otherwise produce false

**find-route/list**: check step

- check each potential step forward (i.e. try each neighbour) with **find-route**
- if any attempt is successful report it
Implementation of **find-route**

;; (find-route orig dest G) finds route from orig to dest in G if it exists
;; find-route: Node Node Graph → (anyof (listof Node) false)

(define (find-route orig dest G)
  (cond [(symbol=? orig dest) (list dest)] ; 1
    [else (local [(define nbrs (neighbours orig G)) ; 2
      (define route (find-route/list nbrs dest G))] ; 3
        (cond [(false? route) false] ; 4
          [else (cons orig route)])))]])) ; 5
find-route

• Line 1: (base case) if the origin and the destination match ⇒ report origin

• Lines 2–5: (mutually) recursive case
  – Line 2: produce \textit{nbrs}, a list of neighbours of \textit{orig}
  – Line 3: apply \textit{find-route/list} to the list of neighbours, \textit{nbrs}
  – Line 4: if none of \textit{bnrs} is part of the route, produce \textit{false}
  – Line 5: otherwise add \textit{orig} to the route (list of nodes)
Implementation of \texttt{find-route/list}

\texttt{;; (find-route/list los dest G) produces route from an element of los to dest in G, if one exists}

\texttt{;; find-route/list: (listof Node) Node Graph \rightarrow (anyof (listof Node) false)}

\texttt{(define (find-route/list los dest G)}
\begin{verbatim}
  (cond [(empty? los) false] ; 6
        [else (local [(define route (find-route (first los) dest G))] ; 7
                     (cond [(false? route) ; 8
                            (find-route/list (rest los) dest G)] ; 9
                            [else route]))])) ;10
\end{verbatim}
\texttt{(define (find-route/list los dest G)}
\end{verbatim}

CS 135 Fall 2019 15: Graphs 20
find-route/list

- Line 6: (base case) no out-neighbours $\Rightarrow$ report false
- Line 7: (mutually recursive case) apply find-route to the first out-neighbours
- Lines 8–9: (recursive case) if there is no route from first node in los (line 8) then backtrack and try the rest of the neighbours (rest los) (line 9)
- Line 10: (terminate) if there is a route, report it (i.e. stop looking)
Trace trees

If we wish to trace find-route, trying to do a linear trace would be very long, both in terms of steps and the size of each step. Our traces also are listed as a linear sequence of steps, but the computation in find-route is better visualized as a tree.

**Key Point:** 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.
Trace tree for **find-route**

(find-route 'A)
  (find-route/list '(C D E))
    (find-route 'C)
      (find-route/list empty)
    (find-route 'D)
      (find-route/list '(F J))
        (find-route 'F)
          (find-route/list '(K H))
            (find-route 'K)
              (find-route/list empty)
        (find-route 'J)
          (find-route/list '(H))
            (find-route 'H)
              (find-route/list empty)
    (find-route 'E)
      (find-route/list '(K))
        (find-route 'K)
          (find-route/list empty)

**find-route v** applies **find-route/list** to the out-neighbours of v

**find-route/list lov** applies **find-route** on each v in lov
Backtracking in implicit graphs

The only places where real computation is done on the graph is in comparing the origin to the destination and in the neighbours function.

*Key Idea:* 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.
Other uses of backtracking

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).

*Key Idea:* 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.
Backtracking in implicit graphs

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 in AI

Backtracking forms the basis of many artificial intelligence programs.

These programs generally add heuristics to determine which neighbour to explore first, or which ones to skip because they appear unpromising.
Termination of **find-route** (no cycles)

In a directed acyclic graph, any route with a given origin will recurse on its (finite number) of neighbours by way of `find-route/list`.

**Key Point:** The origin will never appear in this call or any subsequent calls to `find-route`: if it did, we would have a cycle in our DAG.

Thus, the origin will never be explored in any later call, and thus the subproblem is smaller. Eventually, we will reach a subproblem of size 0 (when all reachable nodes are treated as the origin).

Thus `find-route` always terminates for directed acyclic graphs.
Non-termination of **find-route** (cycles)

It is possible that **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?
Non-termination of **find-route** (cycles)

(find-route 'A)
(find-route/list '(B))
(find-route 'B)
(find-route/list '(C))
(find-route 'C)
(find-route/list '(A))
(find-route 'A)
(find-route/list '(B))
...

**find-route** will never terminate, even if a path from A to D was added, e.g. '(C (A D)).
Improving \textbf{find-route}

\textit{For exam: Just understand broad ideas not implementation details.}

\textbf{Approach:} We can \textit{use accumulative recursion} to solve the problem of \texttt{find-route} possibly not terminating if there are cycles in the graph.

\textit{Key Idea:} 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 \texttt{find-route/list}. 
Overview of `find-route` and `find-route/list` v2.0

Key Idea: create `visited` that accumulates all the nodes visited so far

- **find-route/list**: check step
  - check if step has been `visited` already before checking a step

- **find-route**: take step
  - add current node, `orig` to list of `visited` nodes
;; find-route/list:
;; (listof Node) Node Graph (listof Node) → (anyof (listof Node) false)
(define (find-route/list los dest G visited)
  (cond [(empty? los) false] ; 6
        [(member? (first los) visited) ; 6a
         (find-route/list (rest los) dest G visited)] ; 6b
        [else (local [(define route (find-route/acc (first los)
                                               dest G visited)))] ; 7*
          (cond [(false? route) ; 8
                 (find-route/list (rest los) dest G visited)] ; 9*
                [else route]))])) ;10
Avoiding cycles: find-route/list v2.0

Key Point: This version of find-route/list is very similar to the old version except that it now includes an additional parameter visited to track which nodes have been visited. Changes include ...

- Line 6a, b: if (first los) is a member of visited (i.e. it has already been visited) then skip over this node (i.e. don’t apply find-route/acc to it) just recursive consider the rest of los.

- Line 7*: find-route/acc now includes visited as a argument.

- Line 9*: find-route/list now includes visited as a argument.
Avoiding cycles: find-route/list v2.0

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.
Avoiding cycles: find-route/acc v2.0

;; find-route/acc:

;; Node Node Graph (listof Node) → (anyof (listof Node) false)

(define (find-route/acc orig dest G visited)
  (cond [(symbol=? orig dest) (list dest)] ; 1
    [else (local [(define nbrs (neighbours orig G)) ; 2
                  (define route (find-route/list nbrs dest G ; 3*
                                  (cons orig visited)))]]
      (cons orig visited))]
  (cond [(false? route) false] ; 4
    [else (cons orig route)]))) ; 5
Avoiding cycles: find-route/acc v2.0

This version of find-route/acc is very similar to the old version except that it now includes an additional parameter visited to track which nodes have been visited. Changes include ...

- Line 3*: file-route/list is applied as before, but now orig is consed onto visited to get a longer list of visited nodes.

Notice on slide 35 that each time find-route/acc gets applied visited (the second argument in the diagram) will get longer by one node, the first argument of find-route/acc.

As before, dest and G are not included as arguments on slide 35.
Revisiting our example

```
A B C D E F K H J

CS 135 Fall 2019 15: Graphs
```
Note that the value of the accumulator in \texttt{find-route/list} is always the reverse of the path from A to the current origin (first argument).
Dealing with cycles

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 function terminates.* The accumulator ensures that the depth of recursion is no greater than the number of nodes in the graph, so `find-route` terminates.
Dealing with cycles

(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
Dealing with inefficiency

*For exam: Just understand broad ideas not implementation details.*

In practice, we would write a wrapper function for users which would avoid their having to specify the initial value of the accumulator.

*Key Point:* 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.
Dealing with inefficiency

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.

For each diamond you have a choice 1) take the top pair of edges or 2) take the bottom pair of edges.

That is 2 possibilities for the first diamond, 2 for the second, etc. In total, $2^d$ different combinations of choices.
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.
Tracking visited 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)`.

**Key Idea:** To avoid revisiting nodes, the failed computation should pass the list of nodes it has seen on to the next computation.

**Key Change** 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).
Remembering what the list of nodes represents

We will make a new type that will store both

1. the *list of nodes* (either those nodes which have been visited, or the nodes along the successful path)

2. a *boolean value* indicating what the list of nodes represents.

```
(define-struct routepair (valid-path? nodes))
;; a RoutePair is a (make-routepair Bool (listof Node))
```
Overview of **find-route** and **find-route/list** v3.0

Key Idea: create a struct `routepair` which is a copy the nodes visited from previous attempt (and marks it as a valid route or not).

**find-route/list**: check step

- create `route` a list of all nodes visited
- if step is unsuccessful, use `route` as a list of visited node

**find-route**: take step

- if successful, create a new valid `route`
- otherwise add the current node, `orig` to the unvalid `route`
;; find-route/list: ... → RoutePair
(define (find-route/list los dest G visited)
  (cond [(empty? los) (make-routepair false 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 [(not (routepair-valid-path? route))
                      (find-route/list (rest los) dest G
                          (routepair-nodes route))]
                        [else route]))]))
Improving efficiency: find-route/list v3.0

The changes involve using a routepair rather than using visited nodes or false if no path is found.

- Underline 1: rather than produce false, produce (make-routepair false visited).

- Underline 2: rather than test if route is false, test if the valid-path? field of make-routepair is false.

- Underline 3: rather than pass visited to find-route/list, pass (routepair-nodes route).
(define (find-route/acc orig dest G visited)
  (cond [(symbol=? orig dest) (make-routepair true (list dest))]
        [else (local [(define nbrs (neighbours orig G))
                       (define route (find-route/list nbrs dest G
                                      (cons orig visited)))]
                  (cond [(not (routepair-valid-path? route)) route]
                        [else (make-routepair true
                                  (cons orig
                                    (routepair-nodes route)))])]))]}
Improving efficiency: find-route/acc v3.0

Again, the changes involve using a routepair rather than using visited nodes or false if no path is found.

- Underline 1: For the base case, rather than produce (list orig), produce (make-routepair true (list orig)).

- Underline 2: rather than test if route is false, test if the valid-path? field of make-routepair is false.

- Underline 3: rather than cons orig to visited, cons it to the nodes field of the routepair structure.
The wrapper function \texttt{find-route}

\begin{verbatim}
;; 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 [(routepair-valid-path? route) (routepair-nodes route)]
          [else false])))
\end{verbatim}
Summary of changes

With these changes, find-route runs much faster on the diamond graph.

In future courses we will see how to make find-route even more efficient and 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.
Module 15 Summary

Graphs

1. A directed graph is a set of **vertices** and **edges**. [2]
2. An edge is represented as an ordered pair of vertices. [2]
3. For an edge \((v, w)\), \(w\) is an **out-neighbour** of \(v\), and \(v\) is an **in-neighbour** of \(w\). [5]
4. A **route** is a sequence of vertices \(v_1, v_2, \ldots, v_k\) where 
   \((v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)\) are all edges. [5]
5. A **cycle** is a route that starts and ends at the same vertex. [5]
6. A **directed acyclic graph** or **DAG** is a directed graph without any cycles. [5]
Module 15 Summary

Adjacency Lists and Backtracking

7. An **adjacency list** is a list of all the vertices in a graph with their out-neighbours. [6]

8. (find-route orig dest G) finds a route in graph G from orig to dest. [10]

9. (neighbours v G) produces the list of out-neighbours of v. [10]

10. A route from orig to dest must pass through one of the out-neighbours of orig. [13-14]

11. A **backtracking algorithm** would systematically consider each out-neighbour of orig. [15-16]
Module 15 Summary

Backtracking and Route Finding

12. `find-route` applies finds the out-neighbours of `orig` and applies `find-route/list` to these neighbours, moving one step forward. [17-22]

13. `find-route/list` applies `find-route` to each node in a list of nodes. [17-22]

14. Backtracing can be used on implicit graphs where each node represents a partial solutions and out-neighbours represent the next step (e.g. in game playing). [23-26]
Module 15 Summary
Improvement #1: Dealing with Cycles using visited
15. The previous version of find-route and find-route/list will not terminate if the graph has a cycle. [27-29]
16. These two functions can be improved by using an accumulator visited which tracks which nodes have already be visited and only apply find-route to those who have not been visited yet. [30-37]
17. The new versions of the functions do not perform well in the case of diamonds because there can be an exponential number of different paths from the origin. [38-39]
Module 15 Summary

Improvement #2: Efficiency Considerations using routepair

18. To avoid revisiting nodes, the failed computation should pass the list of nodes it has seen on to the next computation. [40-41]

19. This information is represent in a structure routepair that tracks either a valid route, or the list of nodes that have been visited if the route failed. [42-45]

20. Efficiency is important and will be discussed in future CS courses. [46]
Computing history

Readings: None

Topics:

• Early history [2–5]
• Challenges from Hilbert [6–11]
• Leading up to functional programming (Church) [12–19]
• Leading up to imperative programming (Turing) [13–26]
• “Modern” Computers and Programming Languages [27–38]
• Course summary
The dawn of computation

Babylonian cuneiform circa 2000 B.C.

Many of the surviving tablets track treasury inventories, expenditures, and tax records.

(Photo by Matt Neale)
Early computers and algorithms

The word *computer* originally meant a human being performing computation.

Isaac Newton (1643-1727) hired a *computer* to help him with his computations.

First non-trivial *algorithm*: Euclid’s GCD algorithm circa 300 B.C.

The term *algorithm* comes from the last name of Abu Ja’far Muhammad ibn Musa Al-Khwarizmi’s who wrote books on algebra and arithmetic computation, circa 800 A.D.
Charles Babbage (1791-1871)

Difference Engine (1819)

Analytical Engine (1834)

Mechanical computation for military applications

**Key Point:** The specification of computational operations (using punched cards) was separated from their execution.

Babbage’s designs were technically too ambitious
Ada Augusta Byron (a.k.a. Ada Lovelace) (1815-1852)

Assisted Babbage in explaining and promoting his ideas

*Key Point:* Wrote articles describing the operation and use (programming) 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?

*Key Point*: Wanted to formalize all of mathematics.
The meaning of proof

**Axiom:** \( \forall n : n + 0 = n \).

**Math statement:** “The square of any even number is even.”

**Formula:** \( \forall n (\exists k : n = k + k \Rightarrow \exists m : m + m = n \ast n) \)

**Proof:** Finite sequence of axioms (basic true statements) and derivations of new true statements (e.g. \( \phi \) and \( \phi \rightarrow \sigma \) yield \( \sigma \)).

**Theorem:** A mathematical statement \( \sigma \) together with a proof deriving \( \sigma \) within a given system of axioms and der
Hilbert’s questions about mathematics (1920’s)

Key Point: Hilbert asked fundamental questions about mathematics and proofs.

1. Is mathematics complete? Meaning: for any formula $\phi$, if $\phi$ is true, then $\phi$ is provable.

2. Is mathematics consistent? Meaning: for any formula $\phi$, there aren’t proofs of both $\phi$ and $\neg\phi$.

3. Key Question: Is there a procedure to, given a formula $\phi$, produce a proof of $\phi$, 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 $\phi$ represented by a number $n$ that says “The formula represented by $n$ is not provable.”

The formula $\phi$ cannot be false, so it must be true but not provable.
What remained of Hilbert’s questions

**Remaining Question:** Is there a procedure which, given a formula $\phi$, either proves $\phi$, shows it false, or correctly concludes $\phi$ 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 Hilbert’s question.

With his student Kleene, created notation to describe functions on the natural numbers.
Church and Kleene’s notation: origins of $\lambda$

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? $\Lambda x$.

Too much like the symbol for logical AND: $\land$.

Perhaps a lower-case lambda? $\lambda x$. 
The lambda calculus: function notation

The function that added 2 to its argument would be represented by \( \lambda x. x + 2 \) which today would be written as \( f(x) = x + 2 \).

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).
The lambda calculus: limitations

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?
Notations for numbers (Cantor-style)

0 ≡ ∅ or {} (the empty set)

1 ≡ {∅}

2 ≡ {{∅}, ∅}

In general, \( n \) is represented by the set containing the sets representing \( n - 1, n - 2, \ldots, 0 \).

Key Point: 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)

$1 \equiv \lambda f.\lambda x.fx$ (the function which, when given as argument a function $f$, returns the same function).

$2 \equiv \lambda f.\lambda x.f(fx)$ (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.

**Key Point:** This is the way that arithmetic can be built up from function applications.
Lambda Calculus: creating other functions

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.

**Key Point:** The lambda calculus is a general model of computation. It comes from set theory ⇒ natural numbers ⇒ functions on natural numbers.

**Note:** You do not need to know the details.
Church’s proof

Key Achievement: 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

Turing imagined a model of computation that is very similar (in part inspired) modern computers.

The machine (now called a Turing Machine) had

- finite state control (e.g. today’s processors)
- unbounded storage tape (the equivalent of today’s RAM).
Turing’s proof (1936): asking does it halt?

A “Turing machine” is equivalent to a Racket function, i.e. it consumes input and produces a result.

Turing assumed you could implement a machine (a function), $(\text{halt? m1 i})$ that consumed (1) a machine $m1$ and (2) an input $i$, it would determine if the $m1$ would halt when it consumed $i$.

He could then use $\text{halt?}$ to implement a second function that would tell if a machine would halt or not when fed its own description as an input, i.e. $(\text{define (halt-on-myself? m2) (halt? m2 m2)})$
Using `halts-on-myself?`, one can define a machine (i.e. function) that acts on this information.

\[
\text{(define (m3 m2)}
\text{  (cond [(halts-on-myself? m2) (forever 1)]}
\text{    [else true])})
\]

`m3` uses `halts-on-myself?` to see if `m2` represents a machine which halts when fed its own description.

If so, `m3` runs forever; otherwise, it halts.

When happens with `(m3 m3)`, i.e. `m3` consuming a copy of itself.

It creates a contradiction: `m3` halts iff `m3` does not halt.
So the first machine \textit{halts?} cannot exist.

Turing’s proof also demonstrates the undecidability of proving formulae.

\textbf{Key Point:} You do not have to understand this proof in this course.
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: other contributions

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)
John von Neumann and EDVAC

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.

**Key Point:** The EDVAC contain many features of current computers: random-access memory, CPU, fetch-execute loop, stored program control.

Lacking: support for recursion (unlike Turing’s UK designs)
Grace Murray Hopper (1906-1992)

*Key Contributions:* Wrote the first *compiler*, defined the first *English-like data processing language* (early 1950’s) whose ideas later folded into COBOL (1959).
John Backus and 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
```
John Backus and FORTRAN

*Key Contribution:* In 1957 John Backus designed FORTRAN which 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 criticized 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, COBOL and Lisp

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 and Functional Programming

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

Lisp was a forerunner to Racket. It introduced many of the ideas you see in Racket today.

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 (i.e. first) and cdr (i.e. rest) 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.
Use of Lisp

*Key Point:* 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.
Computing Power in the 1960

• In 1961 (after two short term leases) U Waterloo obtained an IBM 1620.
  – It had 20 KB of memory.
  – It had 2 MB hard drive.
  – It could add or subtract two 5 digit numbers as a rate of 1,780/second.
  – It could multiply two 5 digit numbers a rate of 200/second.
  – This would average out to about 360 operates per second if the mix was 50% multiplication.
Scheme: a descendant of Lisp

Starting about 1976, Carl Hewitt, Gerald Sussman, Guy Steele, and others created a series of research languages called Planner, Conniver, and Schemer (except that “Schemer” was too long for their computer’s filesystem, so it got shortened to “Scheme”).

Because of its simplicity, research groups at other universities began using Scheme to study programming languages.

Sussman, together with colleague Hal Abelson, started using Scheme in the undergraduate program at MIT. Their textbook, “Structure and Interpretation of Computer Programs” (SICP) is considered a classic.
Scheme’s descendant: Racket

The authors of the HtDP textbook developed an extension of Scheme and its learning environment 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 it diverged further from Sussman and Steele’s Scheme, they renamed their language *Racket* in 2010.
Goals of this module

You should understand that important computing concepts pre-date electronic computers.

You should understand, at a high level, the contributions of pioneers such as Babbage, Ada Augusta Byron, Hilbert, Church, Turing, Gödel, and others.

You should understand the origins of functional programming in Church’s work and the origins of imperative programming in Turing’s work.
Summing up CS 135

Key Point: With only a few language constructs (define, cond, define-struct, cons, local, lambda) we have described and implemented ideas from introductory computer science.

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 a 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 with additional themes and a new programming language using a different paradigm.
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?
Looking ahead to CS 136

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?
Looking ahead to CS 136

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.
Module 16 Summary

Dawn of Computing

1. Computation goes back to some of the earliest writing. The earliest computers were people and the earliest algorithms were descriptions of how they could arrive at the result. [2-3]

2. Charles Babbage described the first device that was designed for general compututation where the “program” was specified using punched cards. [4]

3. Ada Augusta Byron is considered by many to be the first computer scientist. She wrote articles describing algorithms for—and explaining how to use—Babbage’s Analytic Engine. [5]
Module 16 Summary

Limits of Mathematics

4. 1900: David Hilbert asked about the completeness and consistency of mathematics and if a procedure for producing proofs is possible. [6-8]

5. 1929-30: Kurt Gödel answered the first two questions: powerful systems are not complete and consistency cannot be proved within the system. [9-10]

6. 1936: Alonzo Church introduced λ-calculus as a formal model of computation and showed that there was no computational method to tell if two λ expressions were equivalent. [12-19]
Module 16 Summary

Computers and Programming Languages

7. 1936: Alan Turing came up with a simpler model of computation (the Turing machine) and showed there is no program that can always determine of another program will halt or not. [20-25]

8. 1946: John von Neumann wrote an influential article about the EDVAC which has many features of a modern computer, called the von Neumann architecture. [26-27]

9. Grace Murray Hopper wrote the first compiler and an English-like programming language which became the basis of COBOL (used in commercial and business applications). [28]
Module 16 Summary

Programming Languages

10. 1957: John Backus created FORTRAN the dominant language for numeric and scientific computation. [29-30]

11. The ideas of Turing, von Neumann, Hopper, and Backus greatly influenced the design of computers. [31]

12. 1958: John McCarthy created Lisp (LISt Processor) taking ideas from $\lambda$-calculus. It became the dominant language in Artificial Intelligence. [32-36]
Module 16 Summary

Programming Languages and Programming Issues

13. Adaptations of Lisp to make it easier to learn in include Scheme (by Gary Sussman and Guy Steele in the 1970s) and then Racket (by Felleisen, Findler, Flatt, Krishnamurthi in the 1990s). [37-38]

14. Looking ahead to CS136 (and beyond) we will consider such issues as scalability (larger data sets), organizing large programs, efficiency and dealing with errors. [42-44]
Preparing for the final

- I will make a complete copy of all my slides (in one big file) available on the course website in the next few days.
- We have a Final Exam [official] post in Piazza giving some suggestions on how to study, listing which slides have been skipped, extra office hours before the final, etc.
- We are having help sessions before the final (two in classrooms and live one on the web).
- We will be monitoring and answering questions in Piazza.