Module 5: Lists

Readings: HtDP, Sections 9, 10.

Lists are the main tool used in Racket to work with unbounded data. As with conditional expressions and structures, the data definition for lists leads naturally to the form of code to manipulate lists.

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

A data definition for lists

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

Using ideas from structures, could we have fields first, second, and so on?

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

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

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

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

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

Recursive definitions

A **recursive** definition of a term has one or more base cases and one or more recursive (or **self-referential**) cases. Each recursive case is defined using the term itself.

Example 1: a mathematical expression

Example 2: a list

- Base case: empty list
- Recursive case: the rest of the list

The rest of the list is smaller than the original list.

What is a list?

A **list** is either

- empty or
- (cons f r), where
  - ⋆ f is a value and
  - ⋆ r is a *list*.

The empty list empty is a built-in constant.

The constructor function cons is a built-in function.

f is the first item in the list. r is the rest of the list.

The empty constant

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

Constructing lists

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

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

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

Deconstructing lists

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

$\Rightarrow$ 'a

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

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

Substitution rules:

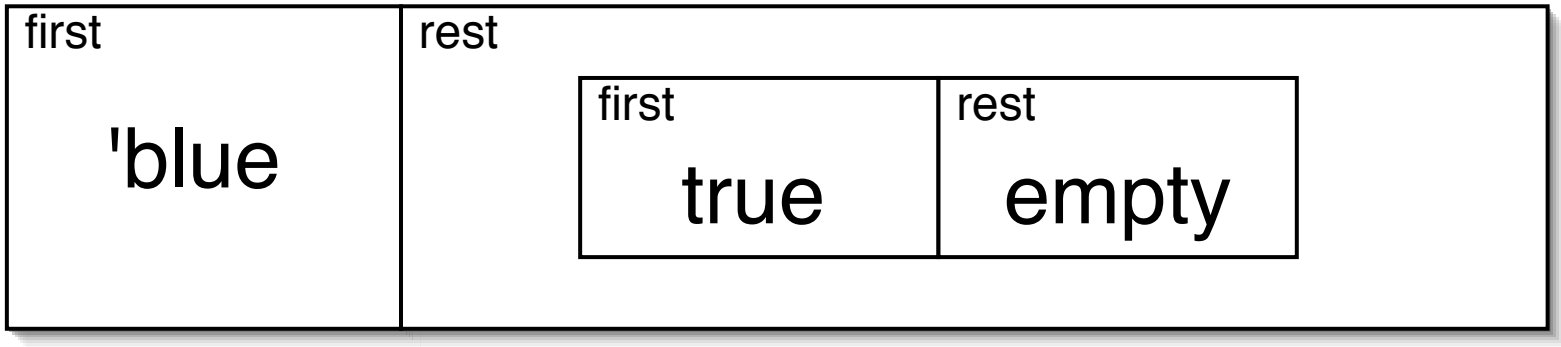
(first (cons elt alist)) $\Rightarrow$ elt

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

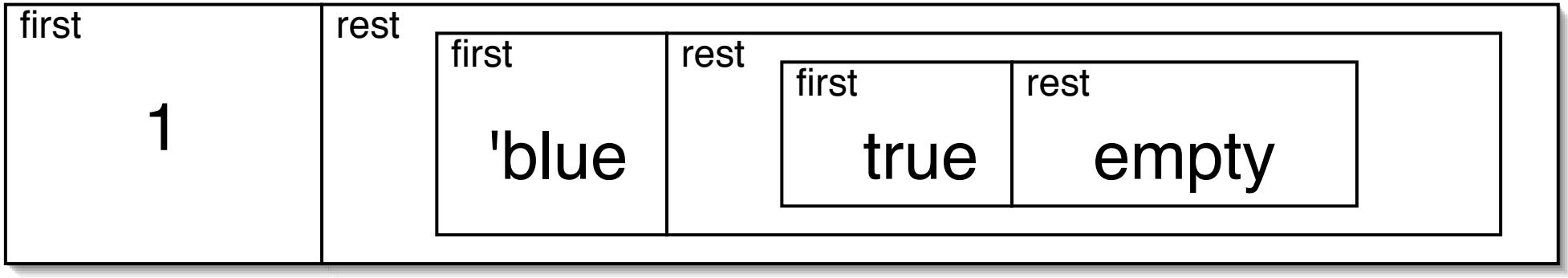
The functions consume nonempty lists only.

Nested boxes visualization

(cons 'blue (cons true empty))

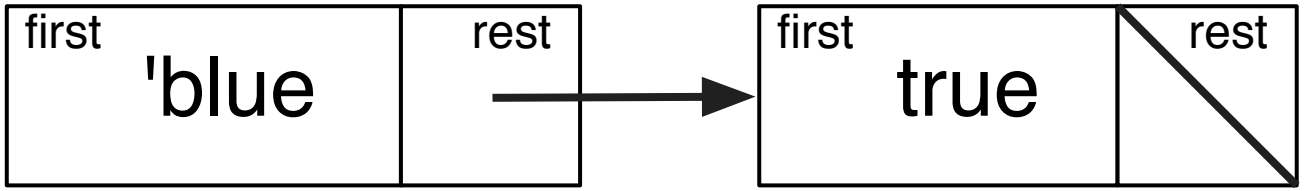


(cons 1 (cons 'blue (cons true empty)))

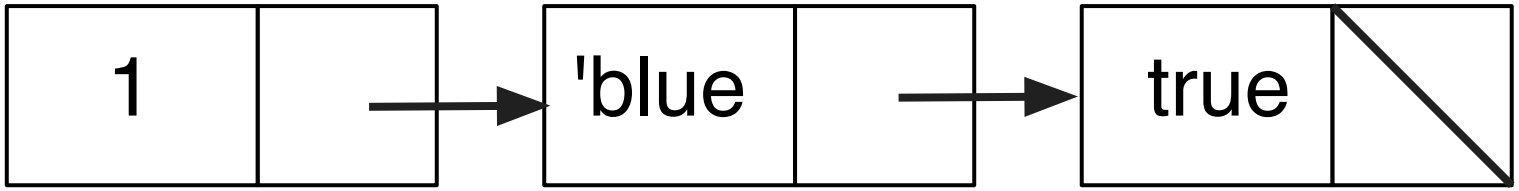


Box-and-pointer visualization

(cons 'blue (cons true empty))



(cons 1 (cons 'blue (cons true empty)))



Extracting values

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

What expression evaluates to:

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

A recursive data definition

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

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

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

Lists are the main data structure in standard Racket.

Which of these are lists of symbols?

empty

(cons 'blue empty)

(cons 'red (cons 'blue empty))

List template

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

```
;; my-los-fun: (listof Sym) → Any
(define (my-los-fun alos)
  (cond
    [(empty? alos) ...]
    [(cons? alos) ...]))
```

The second test can be replaced by else.

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

Developing the template

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

Next idea: since rest alos is also a list of symbols, we can apply my-los-fun to it.

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

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

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

Example: my-length

```
;; (my-length alos) produces the number of symbols in alos.
;; my-length: (listof Sym) → Nat
;; Examples:
(check-expect (my-length empty) 0)
(check-expect (my-length (cons 'a (cons 'b empty))) 2)

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

Tracing my-length

```
(my-length (cons 'a (cons 'b empty)))
⇒ (cond [(empty? (cons 'a (cons 'b empty))) 0]
        [else (+ 1 (my-length (rest (cons 'a (cons 'b empty)))))])
⇒ (cond [false 0]
        [else (+ 1 (my-length (rest (cons 'a (cons 'b empty)))))])
⇒ (cond [else (+ 1 (my-length
                    (rest (cons 'a (cons 'b empty)))))])
⇒ (+ 1 (my-length (rest (cons 'a (cons 'b empty)))))
⇒ (+ 1 (my-length (cons 'b empty)))
```

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

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

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

$\Rightarrow$ (+ 1 (+ 1 (my-length (rest (cons 'b empty)))))

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

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

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

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

$\Rightarrow$ (+ 1 1)

$\Rightarrow$ 2

The trace condensed

(my-length (cons 'a (cons 'b empty)))

$\Rightarrow$ (+ 1 (my-length (cons 'b empty)))

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

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

$\Rightarrow$ 2

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

Condensed traces

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

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

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

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

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

Structural recursion

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

The result is **structural recursion**.

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

You are expected to write structurally recursive code.

Using the templates will ensure that you do so.

Design recipe refinements

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

Do this once per self-referential data type:

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

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

Template: The template follows directly from the data definition.

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

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

Base case clauses will not lead to recursion.

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

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

The (listof Sym) template revisited

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

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

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

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

;; my-lon-fun: (listof Num) → Any
(define (my-lon-fun alon)
  (cond
    [(empty? alon) ...]
    [else ... (first alon) ... (my-lon-fun (rest alon)) ...]))
```

Contracts

There is a difference between types we know about and types that DrRacket knows about.

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

We also use terms in contracts that DrRacket does not know about. These include anyof types, names we have defined in a data definition (like Mp3Info for a structure), and now lists such as (listof Sym) and (listof Num).

listof notation in contracts

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

We'll use (listof X) in contracts, where X may be replaced with any type. Examples: (listof Num), (listof Bool), (listof (anyof Num Sym)), (listof Mp3Info), and (listof Any).

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

Templates

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

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

You will use this template many times!

Templates as generalizations

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

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

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

Use the templates!

Design recipe modifications, continued

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

Body: Use examples to fill in the template.

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

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

Example: count-apples

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

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

Generalizing count-apples

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

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

Extra information in a list function

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

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

Built-in list functions

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

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

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

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

Producing lists from lists

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

For example:

(negate-list empty) $\Rightarrow$ empty

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

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

Building negate-list from template

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

negate-list completed

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

Condensed trace of negate-list

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

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

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

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

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

⇒ (cons −2 (cons 12 empty))

Removing elements from a list

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

Complete singles to produce a list with only the first occurrence of each element. You may use removal.

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

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

Wrapping a function in another function

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

Consider using a wrapper function

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

Strings and characters

Strings are made up of zero or more character values.

In Racket, characters are a separate type.

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

Built-in functions: char<? to compare alphabetical order.

Built-in predicates char-upper-case?.

The course web page has more information about characters.

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

Strings and lists of characters

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

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

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

We use lists of characters in labs, assignments, and exams.

Using wrappers for string functions

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

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

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

Canadianizing a string

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

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

You will write canadianize-list in lab.

Determining portions of a total

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

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

For an empty list, we produce the empty list.

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

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

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

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

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

Nonempty lists

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

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

```
;; (max-list lon) produces the maximum element of lon
;; max-list: (listof Num) → Num
;; requires: lon is nonempty

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

Functions with multiple base cases

Suppose we wish to determine whether a list of numbers has all items the same.

What should the function produce if the list is empty?

What if the list has only one item?

Structures containing lists

Suppose we store the name of a server along with a list of tips collected.

How might we store the information?

```
(define-struct server (name tips))
;; A Server is a (make-server Str (listof Num))
;; requires:
;;      numbers in tips are non-negative
```

We form templates for a server and for a list of numbers.

```
(define (my-server-fun s)
  ... (server-name s) ...
  ... (my-lon-fun (server-tips s)) ... )

(define (my-lon-fun alon)
  (cond
    [(empty? alon) ...]
    [else ... (first alon) ...
          ... (my-lon-fun (rest alon)) ...]))
```

The function big-tips consumes a server s and a number smallest and produces the server formed from s by removing tips smaller than smallest.

```
(define (big-tip-list alon smallest)
  (cond [(empty? alon) empty]
        [(<= smallest (first alon))
         (cons (first alon) (big-tip-list (rest alon) smallest))]
        [else (big-tip-list (rest alon) smallest)]))

(define (big-tips s smallest)
  (make-server (server-name s) (big-tip-list (server-tips s) smallest)))
```

Lists of structures

Suppose we wish to store marks for all the students in a class.

How do we store the information for a single student?

```
(define-struct student (name assts mid final))
;; A Student is a (make-student Str Num Num Num)
;; requires:
;;      assts, mid, final are between 0 and 100
```

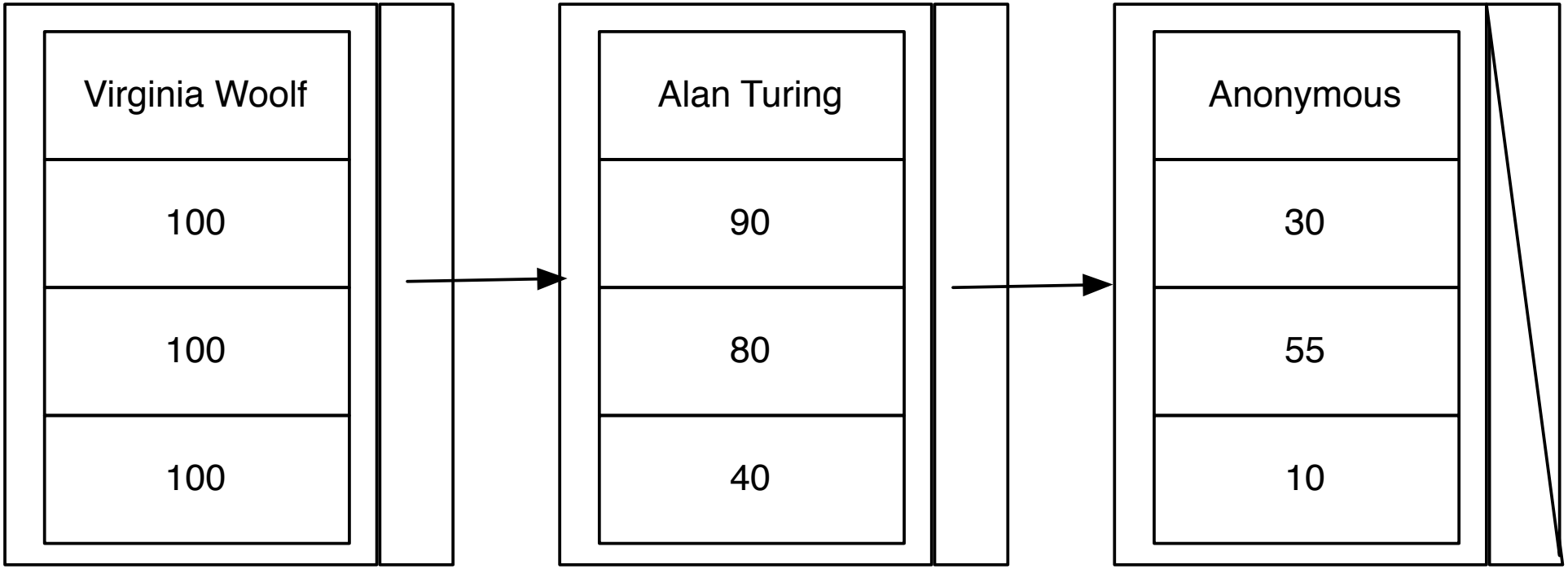
How do we store information for all the students?

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

Example:

```
(cons (make-student "Virginia Woolf" 100 100 100)
   (cons (make-student "Alan Turing" 90 80 40)
      (cons (make-student "Anonymous" 30 55 10) empty)))
```

Visualization



A list of students

```
(define myslist
  (cons (make-student "Virginia Woolf" 100 100 100)
    (cons (make-student "Alan Turing" 90 80 40)
      (cons (make-student "Anonymous" 30 55 10) empty))))
```

What are the values of these expressions?

- `(first myslist)`
- `(rest myslist)`
- `(student-mid (first (rest myslist)))`

The template for a list of students

We first break up a list of students using selectors for lists.

```
;; my-studentlist-fun: (listof Student) → Any
(define (my-studentlist-fun slist)
  (cond
    [(empty? slist) ...]
    [else ... (first slist)... (my-studentlist-fun (rest slist))...]))
```

Since (first slist) is a student, we can refine the template using the selectors for student.

```
;; my-studentlist-fun: (listof Student) → Any
(define (my-studentlist-fun slist)
  (cond
    [(empty? slist) ...]
    [else ... (student-name (first slist)) ...
          ... (student-assts (first slist)) ...
          ... (student-mid (first slist)) ...
          ... (student-final (first slist)) ...
          ... (my-studentlist-fun (rest slist))...]))
```

It may be convenient to define constants for sample data, as we did for Cards.

This can reduce typing and make our examples and tests clearer.

Remember not to cut and paste from the Interactions window to the Definitions window.

```
;; Sample data
(define vw (make-student "Virginia Woolf" 100 100 100))
(define at (make-student "Alan Turing" 90 80 40))
(define an (make-student "Anonymous" 30 55 10))

(define classlist (cons vw (cons at (cons an empty))))
```

The function name-list

```
;; (name-list slist) produces a list of names in slist.
;; name-list: (listof Student) → (listof Str)
(check-expect (name-list classlist)
   (cons "Virginia Woolf" (cons "Alan Turing"
      (cons "Anonymous" empty))))

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

Computing final grades

Suppose we wish to determine final grades for students based on their marks in each course component.

How should we store the information we produce?

```
(define-struct grade (name mark))
;; A Grade is a (make-grade Str Num),
;; requires:
;;      mark is between 0 and 100, inclusive.
```

The function compute-grades

```
;; (compute-grades slist) produces a grade list from slist.
;; compute-grades: (listof Student) → (listof Grade)
;; Examples:
(check-expect (compute-grades empty) empty)
(check-expect (compute-grades classlist)
   (cons (make-grade "Virginia Woolf" 100)
      (cons (make-grade "Alan Turing" 62)
         (cons (make-grade "Anonymous" 27.5) empty))))
```

To enhance readability, we may choose to put the structure selectors in a helper function instead of in the main function.

```
(define (my-student-fun s)
    ... (student-name s) ... (student-assts s) ...
    ... (student-mid s) ... (student-final s) ...)
(define (my-studentlist-fun slist)
  (cond
    [(empty? slist) ...]
    [else ... (my-student-fun (first slist))
          ... (my-studentlist-fun (rest slist))...]))
```

The helper function final-grade

```
;; Constants for use in final-grade
(define assts-weight .20)
(define mid-weight .30)
(define final-weight .50)

;; (final-grade astudent) produces a grade from the astudent, with
;;    20 for assignments, 30 for midterm, and 50 for final
;; final-grade: Student → Grade
;; example:
(check-expect (final-grade vw) (make-grade "Virginia Woolf" 100))
```

```
(define (final-grade astudent)
  (make-grade
   (student-name astudent)
   (+ (* assts-weight (student-assts astudent))
      (* mid-weight (student-mid astudent))
      (* final-weight (student-final astudent)))))
(define (compute-grades slist)
  (cond
    [(empty? slist) empty]
    [else (cons (final-grade (first slist)) (compute-grades (rest slist)))]))
```

Condensed trace of compute-grades

```
(compute-grades myslist) ⇒
(compute-grades
  (cons (make-student "Virginia Woolf" 100 100 100)
    (cons (make-student "Alan Turing" 90 80 40)
      (cons (make-student "Anonymous" 30 55 10) empty)))) ⇒
(cons (make-grade "Virginia Woolf" 100)
  (compute-grades
    (cons (make-student "Alan Turing" 90 80 40)
      (cons (make-student "Anonymous" 30 55 10) empty)))) ⇒
```

```
(cons (make-grade "Virginia Woolf" 100)
  (cons (make-grade "Alan Turing" 62)
    (compute-grades
      (cons (make-student "Anonymous" 30 55 10) empty)))) ⇒
(cons (make-grade "Virginia Woolf" 100)
  (cons (make-grade "Alan Turing" 62)
    (cons (make-grade "Anonymous" 27.5)
      (compute-grades empty)))) ⇒
```

```
(cons (make-grade "Virginia Woolf" 100)
  (cons (make-grade "Alan Turing" 62)
    (cons (make-grade "Anonymous" 27.5) empty)))
```

Goals of this module

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

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

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

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

You should be aware of situations that require wrapper functions.

You should understand how to use (listof . . .) notation in contracts, as well as how to use pre- and post-conditions for restrictions on lists and list values.

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