Module 5: Lists

Readings: HtDP, Sections 9, 10.

Lists are the main tool used in Scheme 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 (allowing us to not know the total number of students ahead of time) and we could use list functions to extract 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 non-empty list of numbers, what remains is a smaller list of numbers.

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: an 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.

Data definition of 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.

Constructing lists

Any non-empty 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:

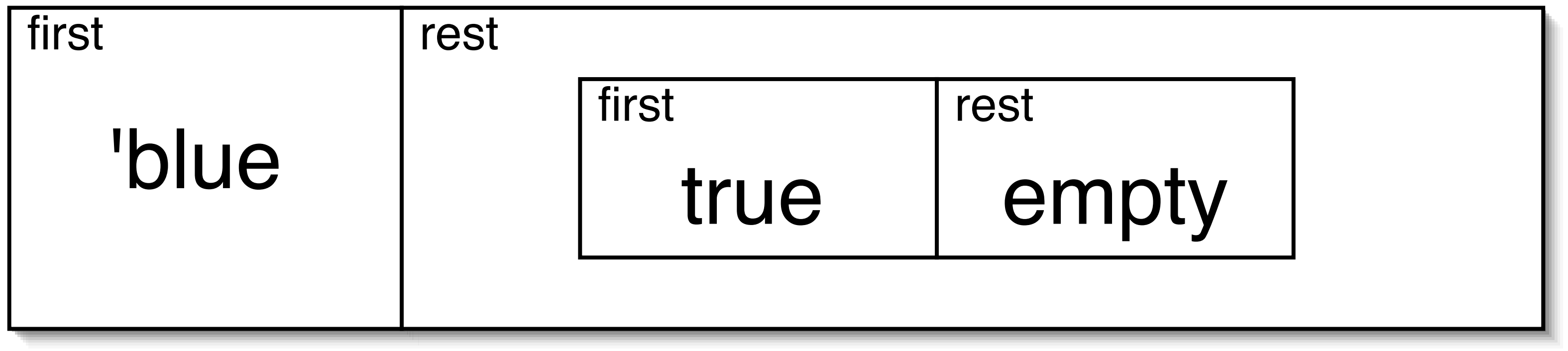
(first (cons elt alist)) $\Rightarrow$ elt

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

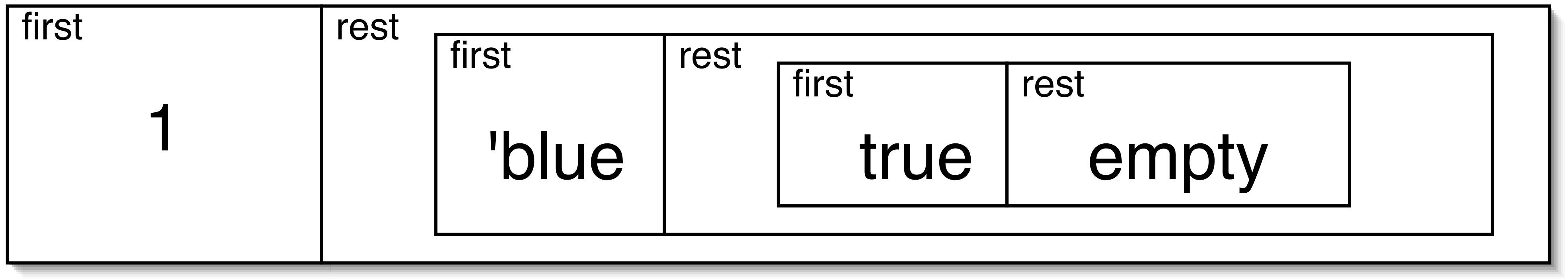
The functions consume non-empty lists only.

Nested boxes visualization

(cons ’blue (cons true empty))

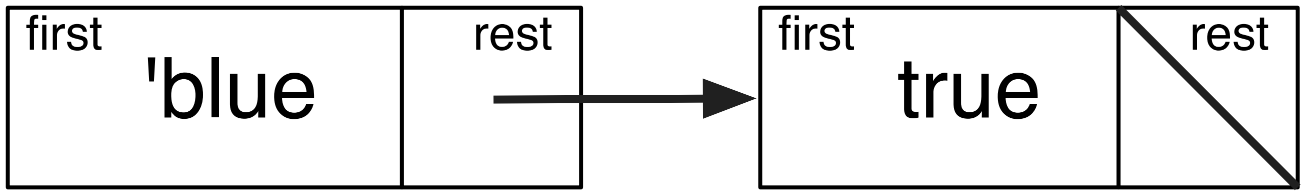


(cons 1 (cons ’blue (cons true empty)))

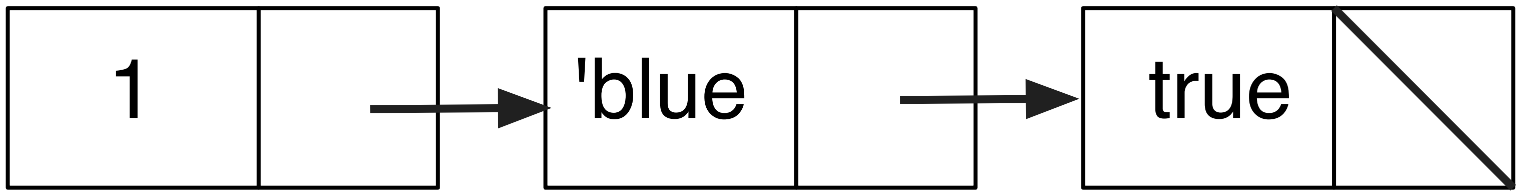


Box-and-pointer visualization

(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 **list of symbols** is either

- empty or
- (cons s alos), where
  - ⋆ s is a symbol and
  - ⋆ alos is a *list of symbols*.

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 symbol) → any
(define (my-los-fun alos)
  (cond
    [(empty? alos) . . . ]
    [(cons? alos) . . . ]))
```

Use the (listof . . .) notation in contracts; the dots can be any type, including any, mixed data, or a structure.

The second test can be replaced by else.

Like in the structure template, we can use selectors to take apart the non-empty list.

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

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

```
;; my-los-fun: (listof symbol) → 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: how-many-symbols

```
;; how-many-symbols: (listof symbol) → nat
;; Produces the number of symbols in alos.
;; Examples: (how-many-symbols empty) ⇒ 0
;; (how-many-symbols (cons 'a (cons 'b empty))) ⇒ 2
(define (how-many-symbols alos)
  (cond
    [(empty? alos) 0]
    [else (+ 1 (how-many-symbols (rest alos)))]))
```

Tracing how-many-symbols

```
(how-many-symbols (cons 'a (cons 'b empty)))
⇒ (cond [(empty? (cons 'a (cons 'b empty))) 0]
     [else (+ 1 (how-many-symbols (rest (cons 'a (cons 'b empty)))))])
⇒ (cond [false 0]
     [else (+ 1 (how-many-symbols (rest (cons 'a (cons 'b empty)))))])
⇒ (cond [else (+ 1 (how-many-symbols
                          (rest (cons 'a (cons 'b empty)))))])
⇒ (+ 1 (how-many-symbols (rest (cons 'a (cons 'b empty)))))
⇒ (+ 1 (how-many-symbols (cons 'b empty)))
⇒ (+ 1 (cond [(empty? (cons 'b empty)) 0]
             [else (+ 1 . . . )]))
⇒ (+ 1 (cond [false 0] [else (+ 1 . . . )]))
⇒ (+ 1 (cond [else (+ 1 . . . )]))
⇒ (+ 1 (+ 1 (how-many-symbols (rest (cons 'b empty)))))
⇒ (+ 1 (+ 1 (how-many-symbols empty)))
⇒ (+ 1 (+ 1 (cond [(empty? empty) 0] [else (+ 1 . . . )])))
⇒ (+ 1 (+ 1 (cond [true 0] [else (+ 1 . . . )])))
⇒ (+ 1 (+ 1 0)) ⇒ (+ 1 1) ⇒ 2
```

The trace condensed

(how-many-symbols (cons 'a (cons 'b empty)))

$\Rightarrow$ (+ 1 (how-many-symbols (cons 'b empty)))

$\Rightarrow$ (+ 1 (+ 1 (how-many-symbols 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 argument;
- 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 not cover in this course.

You are expected to write structurally recursive code.

Using the templates will ensure that you do so.

Design recipe modifications

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

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

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

Template: The template follows directly from the data definition.

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

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

Base case clauses will not lead to recursion.

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

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

We already saw this in the case of union types.

DrRacket only knows about types “empty” and “cons”, as tested by the predicates empty? and cons?.

But we may know, or expect, something more, for instance that the list (cons 3 empty) should be regarded as of type (listof num).

There is nothing specific about symbols in this template.

It can be further generalized to a list template, which doesn't specify the base type.

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

Templates as generalizations

Templates reduce the need to use examples as a basis for writing new code.

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: Exercise all parts of the data definition; for lists, at least one base case and one recursive case.

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 in the list and the result of applying the function to the rest of the list). Then figure out how to combine these values to obtain the value produced by the function.

Design recipe for self-referential data

1. **Data analysis and design**. At least one base case.
2. **Template**. Self-referential leads to recursion, base cases do not.
3. **Contract** and **function header**. Use (listof . . .) notation.
4. **Purpose**.
5. **Examples**. At least one for each case in the data definition.
6. **Body**. Use examples to determine how to put together values provided by the template.
7. **Tests**.
8. Run the program.

Example: count-apples

```
;; count-apples: (listof symbol) → nat
;; Produces the number of occurrences of 'apple in alos.
;; Examples:
;; (count-apples empty) ⇒ 0
;; (count-apples (cons 'apple empty)) ⇒ 1
;; (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: (listof symbol) symbol → nat
;; Produces the number of occurrences of asymbol in alos.
;; Examples: (count-given empty 'pear) ⇒ 0
;; (count-given (cons 'apple empty) 'apple) ⇒ 1
;; (count-given (cons 'pear (cons 'peach empty)) 'pear) ⇒ 1
(define (count-given alos asymbol) . . . )
```

Extra information in a list function

Our generalized function has an extra parameter.

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)... (my-extra-info-list-fun (rest alist) info) ...]))
```

Built-in list functions

A closer look at how-many-symbols reveals that it will work just fine on lists of type (listof any). It is the built-in Scheme 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 you have learned how to write them yourself. Do not use remove-all in assignments, as it is not built in to the version used for testing.

Producing lists from lists

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

Examples:

(negate-list empty) $\Rightarrow$ empty

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

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

negate-list with template

```
;; negate-list: (listof num) → (listof num)
;; Produces a list formed by negating each item in alon.
;; Examples: (negate-list empty) ⇒ empty
;; (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

```
;; negate-list: (listof num) → (listof num)
;; Produces a list formed by negating each item in alon.
;; Examples: (negate-list empty) ⇒ empty
;; (negate-list (cons 2 (cons -12 empty))) ⇒
;; (cons -2 (cons 12 empty))
(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

Caution: This is not a built-in function.

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

What can go wrong?

Suppose we now wish to use remove-all to produce a list with only the first occurrence of each element.

How might we err if we were to ignore the templates?

Attempt 1:

```
(define (singles alon)
  (cond
    [(empty? alon) empty]
    [else (singles (remove-all (first alon) (rest alon)))]))
```

Will this work?

We need to keep the first element.

```
(define (singles alon)
  (cond
    [(empty? alon) empty]
    [else (singles (cons (first alon) (remove-all (first alon) (rest alon))))])
```

Creating singles using the template

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

(define (singles alon)
  (cond
    [(empty? alon) empty]
    [else
       (cons (first alon)(remove-all (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 encounter difficulty in putting together several recursive functions, or
- if you find that in your recursive function you need more parameters than a lab or assignment question specifies.

Strings as lists of characters

Although strings and 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 possibly 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: string → string
;; Produces a string in which each o in str
;; is replaced by ou.
;; Examples: (canadianize ”mold”) ⇒ ”mould”
;; (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.

We use the notation (listof num)[nonempty].

The square brackets are used for other restrictions, such as int[$<$0], num[$>=$0,$<=$1], string[len$>$5].

What is (listof (union string[len$>$0] int[$<$0]))[nonempty]?

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

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 structure (make-server n t) where
;;      n is a string (for the server's name) and
;;      t is a list of non-negative numbers (for tips).
```

If we hadn't already defined a list of numbers, we would need to add it to the data definition.

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

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 structure (make-student n a m f), where
;;      n is a string,
;;      a is a number between 0 and 100,
;;      m is a number between 0 and 100, and
;;      f is a number between 0 and 100.
```

How do we store information for all the students?

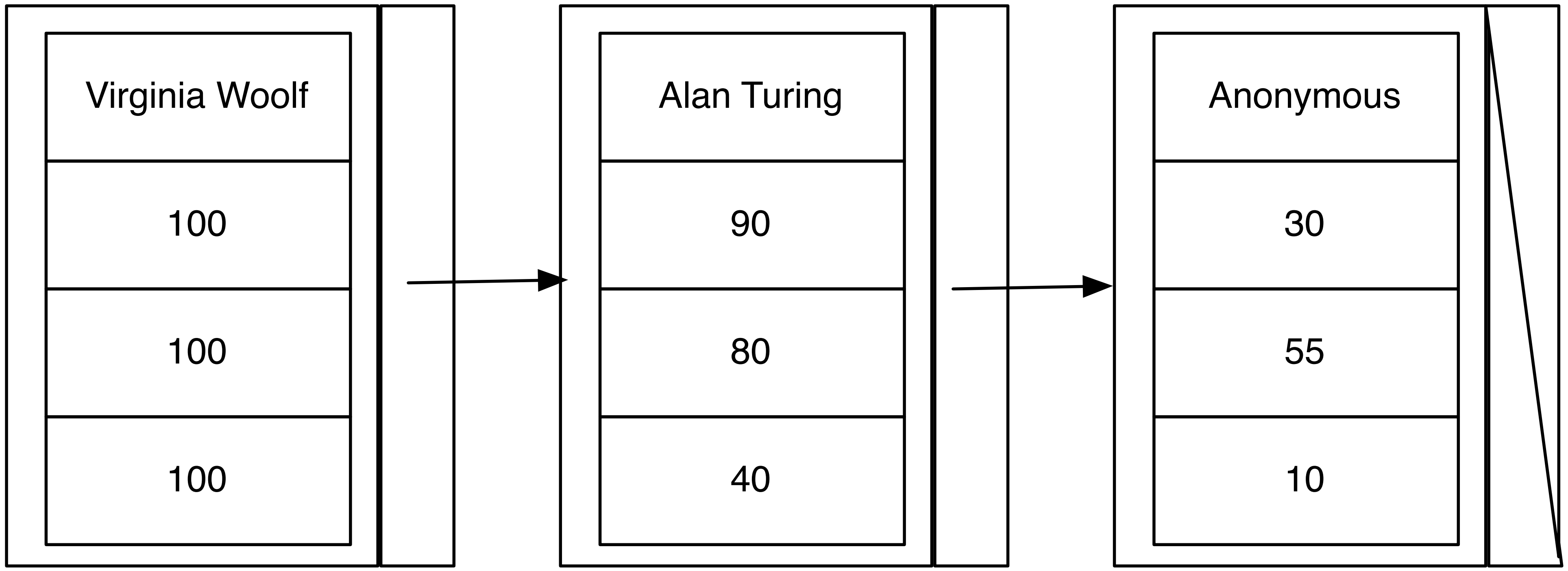
A **list of students** is either

- empty or
- (cons stud slist), where
  - ⋆ stud is a student and
  - ⋆ slist is a *list of students*.

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.

```
(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.

```
(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: (listof student) → (listof string)
;; Produces a list of names in student list slist.
;; Examples: (name-list empty) ⇒ empty
;; (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 structure (make-grade n m), where
;;      n is a string (student’s name) and
;;      m is a number between 0 and 100.
```

Our function will produce a list of structures.

A **list of grades** is either

- empty or
- (cons gr glist), where
  - ⋆ gr is a grade, and
  - ⋆ glist is a *list of grades*.

The function compute-grades

```
;; compute-grades: (listof student) → (listof grade)
;; Produces a grade list from student list slist.
;; Examples: (compute-grades empty) ⇒ empty
;; (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 final-grade instead of in the main function.

We will develop the main function using the first template:

```
(define (my-studentlist-fun slist)
  (cond
    [(empty? slist) . . . ]
    [else . . . (first slist). . . (my-studentlist-fun (rest slist)). . . ]))
(define (compute-grades slist)
  (cond
    [(empty? slist) empty]
    [else (cons (final-grade (first slist)) (compute-grades (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: student → grade
;; Produces a grade from the student astudent, with
;; 20 for assignments, 30 for midterm, and 50 for final
;; Examples:
;; (final-grade vw) ⇒ (make-grade ”Virginia Woolf” 100)
```

```
(define (my-student-fun s)
      . . . (student-name s) . . .
      . . . (student-assts s) . . .
      . . . (student-mid s) . . .
      . . . (student-final s) . . . )
(define (final-grade astudent)
   (make-grade
     (student-name astudent)
     (+ (* assts-weight (student-assts astudent))
          (* mid-weight (student-mid astudent))
          (* final-weight (student-final astudent)))))
```

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)
```

Design recipe for self-referential data

1. **Data analysis and design**. At least one base case.
2. **Template**. Self-referential leads to recursion, base cases do not.
3. **Contract** and **function header**. Use (listof . . .) notation.
4. **Purpose**.
5. **Examples**. At least one base case, at least one recursive case.
6. **Body**. Use examples to determine how to put together values provided by the template.
7. **Tests**.
8. Run the program.

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 . . .) and [nonempty] notation in contracts, as well as other restrictions, and use them in your own functions.

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.