Module 4: Lists

Readings: HtDP, Sections 9, 10.

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

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

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

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

We will write functions that can consume and produce lists.

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

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 3 empty))
```

Deconstructing lists

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

Deconstructing lists

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

$\Rightarrow$ `"a"`

```
(rest (cons "a" (cons "b" (cons "c" 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:

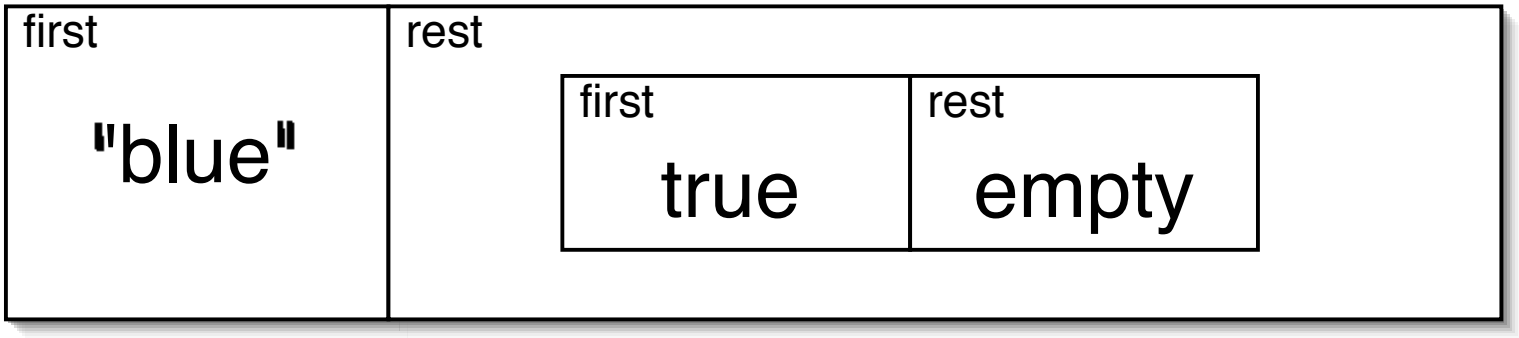
(first (cons elt alist)) $\Rightarrow$ elt

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

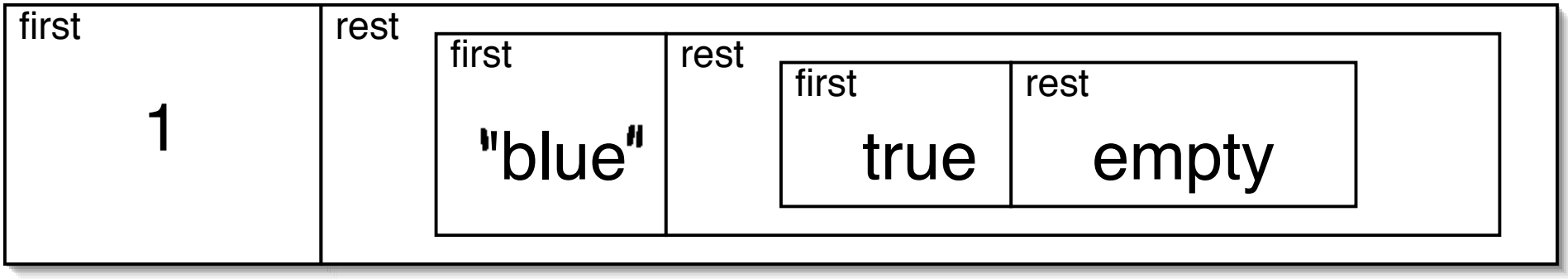
The functions consume nonempty lists only.

Nested boxes visualization

(cons "blue" (cons true empty))

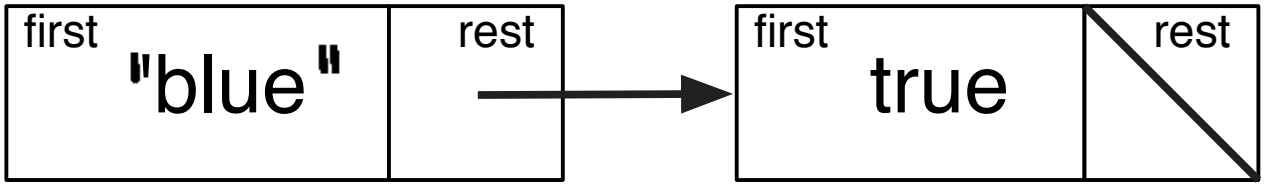


(cons 1 (cons "blue" (cons true empty)))

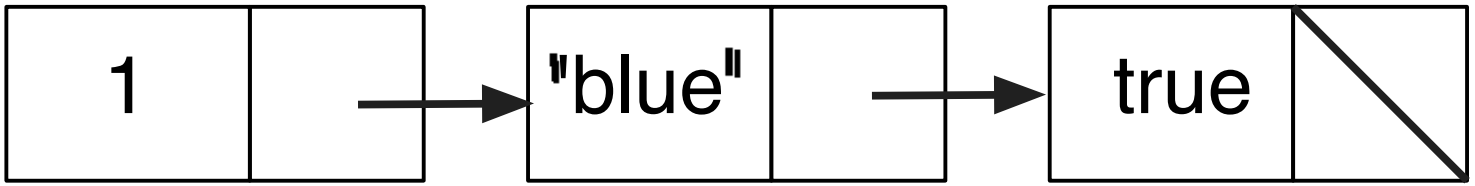


Box-and-pointer visualization

(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 ?

Data-directed design

We often find that the programs we write mirror the structure of the data they are processing. *Data-directed design* captures this connection.

We use the following steps:

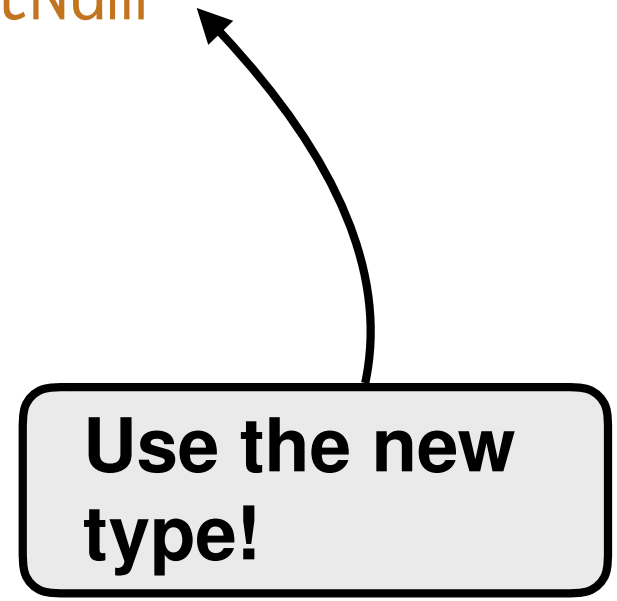
1. **Data Definition:** Introduce a new type name and describe all possible values of that type.
2. **Template:** Work out the high-level structure of typical functions that consume values of the new type.
3. **Function:** Merge back in to the rest of the Design Recipe, refining the template into a real function that operates on values of the new type.

```
;; An ExtNum is one of:
;; * A Num
;; * "undefined"
```

Sample data definition

```
;; An ExtNum is one of:
;; * A Num
;; * "undefined"

;; safe-reciprocal: Num -> ExtNum
(define (safe-reciprocal x)
  (cond
    [(zero? x) "undefined"]
    [else (/ 1 x)]))
```



```
;; An ExtNum is one of:
;; * A Num
;; * "undefined"

;; safe-reciprocal: Num -> ExtNum
(define (safe-reciprocal x)
  (cond
    [(zero? x) "undefined"]
    [else (/ 1 x)]))

;; ext-neg: ExtNum -> ExtNum
(define (ext-neg x)
  (cond
    [(number? x) (- x)]
    [else "undefined"]))
```

```
;; A List is one of:
;; * empty
```

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

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

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

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

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

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

… a self-referential data definition!

Generic list data definition

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

Generic list data definition

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

;; sum-numbers: (listof Num) -> Num
;; longest-word: (listof Str) -> Str
;; another-func: (listof Nat) (listof Str) -> (listof Bool)

Generic list data definition

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


;; sum-numbers: (listof Num) -> Num
;; longest-word: (listof Str) -> Str
;; another-func: (listof Nat) (listof Str) -> (listof Bool)
```

Always replace τ by the most specific type possible in the context.

Constructing a list template

A template is a partially completed function that demonstrates how to consume data of a given type. Let's develop a template for **(listof Int)**.

Constructing a list template

A template is a partially completed function that demonstrates how to consume data of a given type. Let's develop a template for **(listof Int)**.

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

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

;; listof-int-template: (listof Int) → Any
(define (listof-int-template loi)
```

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

;; listof-int-template: (listof Int) → Any
(define (listof-int-template loi)
  (cond
```

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

;; listof-int-template: (listof Int) → Any
(define (listof-int-template loi)
  (cond
    [(empty? loi)
```

empty? produces **true** for the empty list, **false** for any other value.

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

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

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

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

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

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

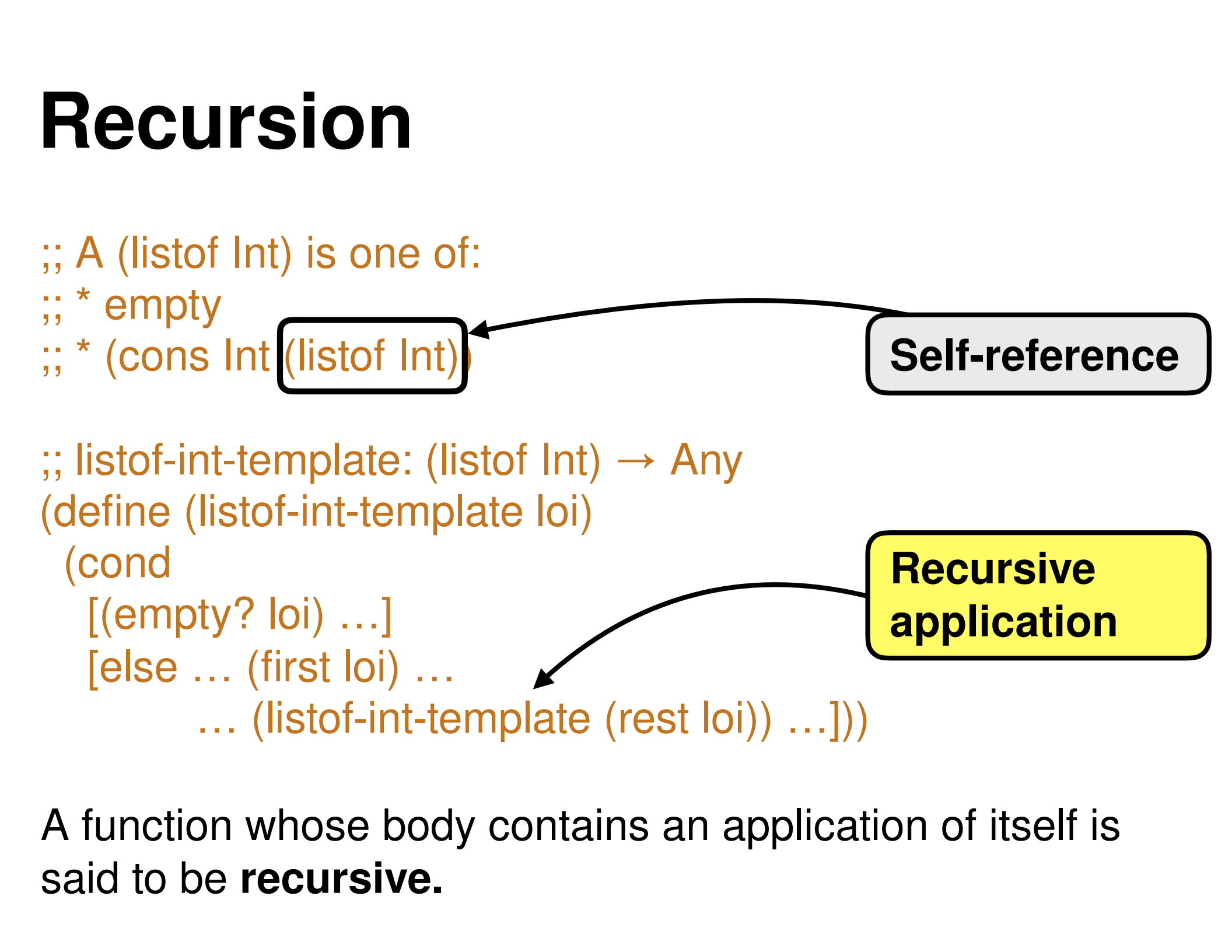
```
;; A (listof Int) is one of:
;; * empty
;; * (cons Int (listof Int))

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

Recursion

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

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



A function whose body contains an application of itself is said to be **recursive.**

Generic list template

From now on, we will use **(listof τ)** freely in contracts, and take the data definition and template below for granted. **You do not need to include them in your code.**

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

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

Structural recursion

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

The result is **structural recursion**.

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

You are expected to write structurally recursive code in CS115.

Using the templates will ensure that you do so.

Using the template

The template is a starting point:

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

The specifics depend on the problem itself.

Example: my-length

;; (my-length aloi) produces the number of integers in aloi.

Tracing my-length

(my-length (cons 3 (cons −2 empty)))

Tracing my-length

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

Tracing my-length

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

Tracing my-length

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

Tracing my-length

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

Tracing my-length

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

```
⇒ (+ 1 (cond [(empty? (cons −2 empty)) 0]
             [else (+ 1 . . . )]))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\Rightarrow$ (+ 1 1)

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

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

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

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

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

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

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

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

$\Rightarrow$ (+ 1 1)

$\Rightarrow$ 2

The trace condensed

(my-length (cons 3 (cons −2 empty)))

⇒ (+ 1 (my-length (cons −2 empty)))

⇒ (+ 1 (+ 1 (my-length empty)))

⇒ (+ 1 (+ 1 0))

⇒ 2

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

Condensed traces

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

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

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

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

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

Design recipe refinements

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

Do this once per self-referential data type:

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

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

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

Template: The template follows directly from the data definition.

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

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

Base case clauses will not lead to recursion.

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

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

Design recipe modifications, continued

Examples/Tests: Exercise all parts of the data definition; for lists, at least one base case 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.