Lists

Readings: HtDP, sections 9 and 10.

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

Introducing lists

Structures are useful for representing a fixed amount of data.

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

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

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

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

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

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

Basic list constructs

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

Example lists

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

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

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

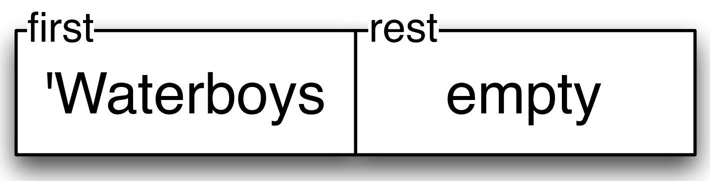
(define clst2alt (cons 'DaCapo (cons 'Waterboys empty))) is just like clst2.

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

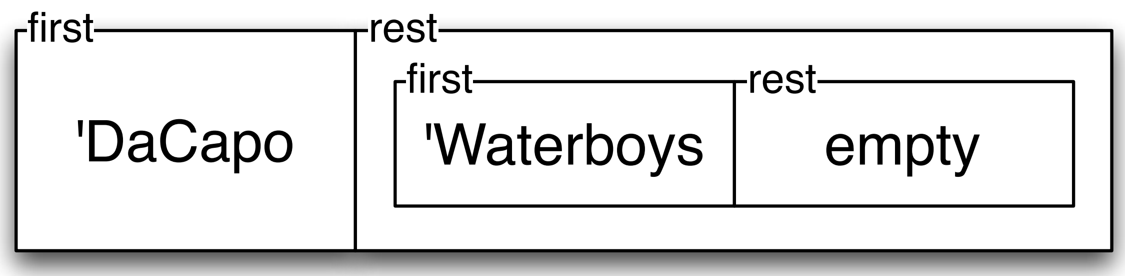
Nested boxes visualization

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

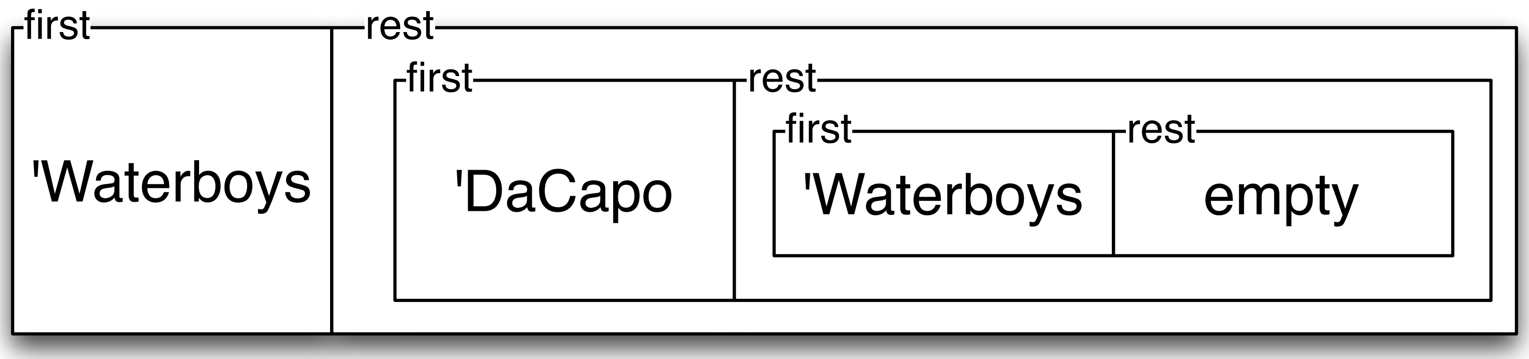
(cons 'Waterboys empty)



(cons 'DaCapo (cons 'Waterboys empty))

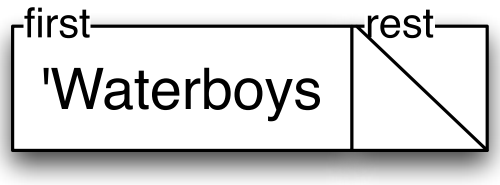


```
(cons 'Waterboys
      (cons 'DaCapo
            (cons 'Waterboys
                  empty)))
```

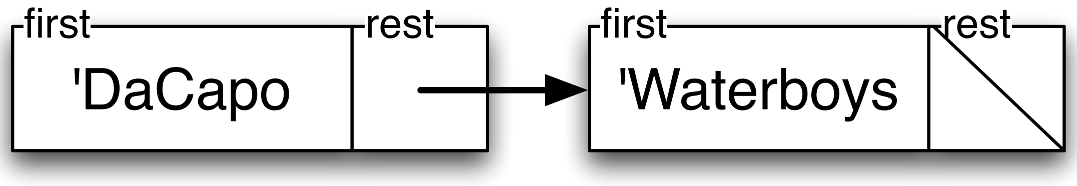


Box-and-pointer visualization

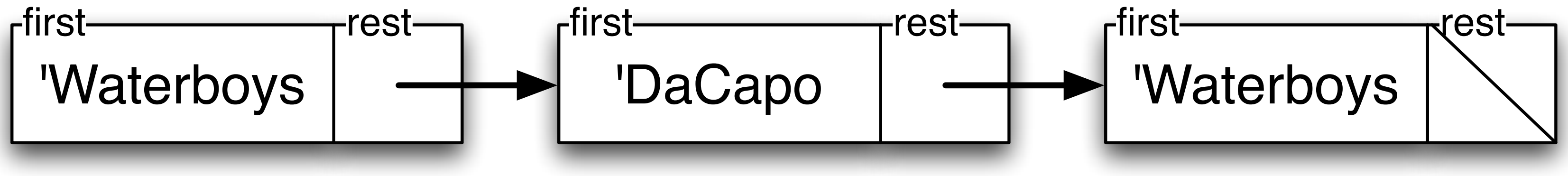
```
(cons 'Waterboys empty)
```



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



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



Extracting values from a list

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

First concert:

(first clst) $\Rightarrow$ 'Waterboys

Concerts after the first:

(rest clst) $\Rightarrow$ (cons 'DaCapo (cons 'Waterboys empty))

Second concert:

(first (rest clst)) $\Rightarrow$ 'DaCapo

Semantics of list functions

(cons a b) is a value.

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

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

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

Functions on lists

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

```
;; (next-concert los) produces the next concert to attend or
;;     false if los is empty
;; next-concert: List-Of-Symbols → (anyof false Sym)
(check-expect (next-concert (cons ’a (cons ’b empty))) ’a)
(check-expect (next-concert empty) false)
(define (next-concert los)
    (cond [(empty? los) false]
              [else (first los)]))
```

```
;; (same-consec? los) determines if next two concerts are the same
;; same-consec?: List-Of-Symbols → Bool
(check-expect (same-consec? (cons 'a (cons 'b empty))) false)
(check-expect (same-consec? (cons 'a (cons 'a empty))) true)
(check-expect (same-consec? (cons 'a empty)) false)

(define (same-consec? los)
  (and (not (empty? los))
       (not (empty? (rest los)))
       (symbol=? (first los) (first (rest los)))))
```

Processing lists

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

We need a data definition and a template. Recall our earlier, informal definition of a list:

- 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.

List data definition

```
;; A List-Of-Symbols is one of:
;; ⋆ empty
;; ⋆ (cons Sym List-Of-Symbols)
```

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.

Template for processing a list of symbols

```
;; my-los-fn: List-Of-Symbols → Any
(define (my-los-fn los)
  (cond [(empty? los) ...]
        [(cons? los) ...]))
```

The second test can be replaced by else.

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

```
;; my-los-fn: List-Of-Symbols → Any
(define (my-los-fn los)
  (cond [(empty? los) . . . ]
        [else (. . . (first los) . . . (rest los) . . . )]))
```

Now we go a step further.

Since (rest los) is a list of symbols, we apply the same computation to it – that is, we apply my-los-fn.

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

```
;; my-los-fn: List-Of-Symbols → Any
(define (my-los-fn los)
   (cond [(empty? los) . . . ]
            [else (. . . (first los) . . .
                            (my-los-fn (rest los)) . . . )]))
```

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

Example: how many concerts?

```
;; (count-concerts los) counts the number of concerts in los
;; count-concerts: List-Of-Symbols → Nat
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons 'a (cons 'b empty))) 2)
(define (count-concerts los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-concerts (rest los)))]))
```

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

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

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

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

Tracing count-concerts

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

⇒ (+ 1 (cond [(empty? (cons 'b empty)) 0][else (+ 1 . . .)]))

⇒ (+ 1 (cond [false 0][else (+ 1 . . .)]))

⇒ (+ 1 (cond [else (+ 1 . . .)]))

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

⇒ (+ 1 (+ 1 (count-concerts empty)))

⇒ (+ 1 (+ 1 (cond [(empty? empty) 0][else (+ 1 . . .)])))

⇒ (+ 1 (+ 1 (cond [true 0][else (+ 1 . . .)])))

⇒ (+ 1 (+ 1 0)) ⇒ (+ 1 1) ⇒ 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.

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: List-Of-Symbols → Nat
;; Examples:
(check-expect (count-waterboys empty) 0)
(check-expect (count-waterboys (cons 'Waterboys empty)) 1)
(check-expect (count-waterboys (cons 'DaCapo
                                     (cons 'U2 empty))) 0)

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

The template may need to be altered if the function requires additional parameters.

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

```
;; count-symbol: Symbol List-Of-Symbols → Nat
(define (count-symbol s los) . . . )
```

The recursive call will be (count-symbol s (rest los)).

Structural recursion

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

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

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

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

Use the templates.

Design recipe refinements

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

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

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

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

Template: The template follows directly from the data definition.

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

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

Base case clauses will not lead to recursion.

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

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

The List-Of-Symbols template revisited

```
;; A List-Of-Symbols is one of:
;; ⋆ empty
;; ⋆ (cons Sym List-Of-Symbols)

;; my-los-fn: List-Of-Symbols → Any
(define (my-los-fn los)
  (cond [(empty? los) . . . ]
        [else (. . . (first los). . . (my-los-fn (rest los)). . . )]))
```

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

```
;; A List-Of-Nums is one of:
;; ⋆ empty
;; ⋆ (cons Num List-Of-Nums)

;; my-lon-fn: List-Of-Nums ⟶ Any
(define (my-lon-fn lon)
  (cond [(empty? lon) ...]
        [else (... (first lon)... (my-lon-fn (rest lon))...)]))
```

Contracts

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

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

(listof X) notation in contracts

As we noted, the data definition for List-Of-Symbols and List-Of-Nums is the same except that one uses Sym and the other 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).

Templates

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

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

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

Templates as generalizations

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

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

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

Filling in the templates

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

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

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

(Pure) structural recursion

In (pure) structural recursion, all parameters to the recursive call (or calls, if there are more than one) are either:

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

Useful list functions

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

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

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

Producing lists from lists

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

```
;; (negate-list lon) negates every number in lon
;; negate-list: (listof Num) → (listof Num)
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons −12 empty)))
                    (cons −2 (cons 12 empty)))
```

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

negate-list with template

```
;; (negate-list lon) negates every number in lon
;; 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)) . . . )]))
```

negate-list completed

```
;; (negate-list lon) negates every number in lon
;; 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) empty]
        [else (cons (− (first lon)) (negate-list (rest lon)))]))
```

A condensed trace

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

Nonempty lists

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

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

Contracts for nonempty lists

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

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

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

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

Strings and lists of characters

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

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

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

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

Racket’s notation for the character ‘a’ is #\a.

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

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

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

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

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

Here is the **wrapper function**:

```
;; (count-e str) Counts the number of occurrences of the letter e in str
;; count-e: Str → Nat
;; Examples:
(check-expect (count-e "") 0)
(check-expect (count-e "realize") 2)

(define (count-e str)
  (count-e/list (string→list str)))
```

Lists of structures

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

Consider the following salary record structure:

```
(define-struct sr (name salary))
;; A Salary Record (SR) is a (make-sr Str Num)

;; my-sr-fn: SR → Any
(define (my-sr-fn sr)
    (. . . (sr-name sr) . . .
           (sr-salary sr) . . . ))
```

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

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

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

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

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

The approach you use is a matter of judgment.

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

Example: compute-taxes

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

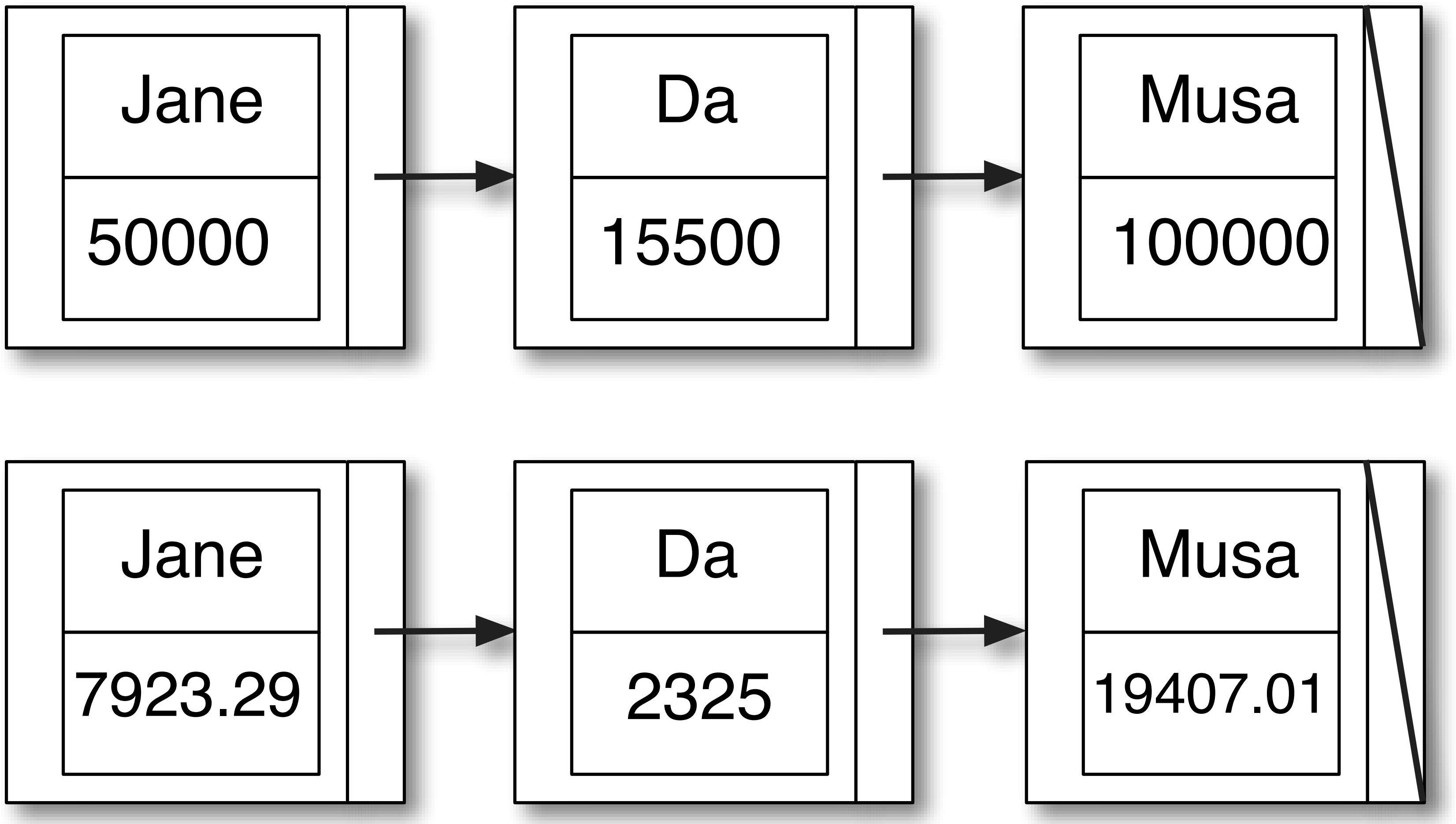
```
(define-struct tr (name tax))
;; A Tax Record (TR) is a (make-tr Str Num)
```

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

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

;; (compute-taxes lst) produces a list of tax records,
;;     one for each salary record in lst
;; compute-taxes: (listof SR) → (listof TR)
;; Example:
(check-expect (compute-taxes srlst)
   (cons (make-tr "Jane Doe" 7923.29)
   (cons (make-tr "Da Kou" 2325)
   (cons (make-tr "MusaAlKhwarizmi" 19407.01) empty))))
```

Envisioning compute-taxes



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

What do these evaluate to?

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

The function compute-taxes

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

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

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

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

We do this with the helper function sr→tr.

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

```
;; (compute-taxes lst) produces a list of tax records,
;;     one for each salary record in lst
;; compute-taxes: (listof SR) → (listof TR)
(define (compute-taxes lst)
  (cond [(empty? lst) empty]
        [else (cons (sr→tr (first lst))
                    (compute-taxes (rest lst)))]))
```

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

```
;; (sr→tr sr) produces a tax record for sr
;; sr→tr: SR → TR
;; Example:
(check-expect (sr→tr (make-sr "Jane Doe" 50000))
              (make-tr "Jane Doe" 7923.29))

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

A condensed trace

```
(compute-taxes srlst)
⇒ (compute-taxes
     (cons (make-sr "Jane Doe" 50000)
       (cons (make-sr "Da Kou" 15500)
         (cons (make-sr "MusaAlKhwarizmi" 100000) empty))))
⇒ (cons (make-tr "Jane Doe" 7923.29)
     (compute-taxes
       (cons (make-sr "Da Kou" 15500)
         (cons (make-sr "MusaAlKhwarizmi" 100000) empty))))
```

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

Goals of this module

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

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

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

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

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

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