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.

List data definition

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

```scheme
;; A List is one of:
;; • empty
;; • (cons Any list)
```

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

A List can hold any combination of values of different types, which isn’t very useful. We will introduce more specific kinds of lists shortly.

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 "U2" (cons "DaCapo" (cons "DaCapo" "empty))))* is a list with one U2 and two DaCapo concerts.
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)
```

```
| "Waterboys" | empty |
```

```
(cons "DaCapo" (cons "Waterboys" empty))
```

```
<table>
<thead>
<tr>
<th>first</th>
<th>rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DaCapo&quot;</td>
<td>first</td>
</tr>
<tr>
<td>&quot;Waterboys&quot;</td>
<td>rest</td>
</tr>
</tbody>
</table>
```

Box-and-pointer visualization

```
(cons "Waterboys" empty)
```

```
<table>
<thead>
<tr>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Waterboys&quot;</td>
</tr>
</tbody>
</table>
```

```
(cons "DaCapo" (cons "Waterboys" empty))
```

```
<table>
<thead>
<tr>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DaCapo&quot;</td>
</tr>
<tr>
<td>first</td>
</tr>
<tr>
<td>&quot;Waterboys&quot;</td>
</tr>
</tbody>
</table>
```

```
(cons "Waterboys" (cons "DaCapo" (cons "Waterboys" empty)))
```

```
<table>
<thead>
<tr>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Waterboys&quot;</td>
</tr>
<tr>
<td>first</td>
</tr>
<tr>
<td>&quot;DaCapo&quot;</td>
</tr>
<tr>
<td>first</td>
</tr>
<tr>
<td>&quot;Waterboys&quot;</td>
</tr>
</tbody>
</table>
```
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\"

Contracts

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

We also use terms in contracts that DrRacket does not know about. These include \textbf{anyof} types, names we have defined in a data definition (like \textbf{SongInfo} for a structure), and now lists.

\[\text{(listof } X \text{) notation in contracts}\]

We'll use \textit{(listof } X \text{)} in contracts, where \textit{X} may be replaced with any type. For the concert list example in the previous slides, the list contains only strings and has type \textit{(listof Str)}.

Other examples: \textit{(listof Num)}, \textit{(listof Bool)}, \textit{(listof (anyof Num Sym))}, \textit{(listof SongInfo)}, and \textit{(listof Any)}.

Replace \textit{X} with the most specific type available.
**listof X** data definition

We can capture the behaviour of a **listof X** by giving a data definition in terms of the type **X**:

;; A listof X is one of:

;; * empty

;; * (cons X list)

Using this data definition, the generic type **List** given earlier can now be expressed as **listof Any**.

---

**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))** ⇒ **a**, where **a** and **b** are values.

- **(rest (cons a b))** ⇒ **b**, where **a** and **b** are values.

- **(empty? empty)** ⇒ **true**.

- **(empty? a)** ⇒ **false**, where **a** is any Racket value other than empty.

- **(cons? (cons a b))** ⇒ **true**.

- **(cons? a)** ⇒ **false**, where **a** is any Racket value not created using **cons**.
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: (listof Str) → (anyof false Str)
(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?: (listof Str) → 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)))
       (string=? (first los) (first (rest los))))))

Processing lists

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

We build a template from our data definition.

;; A (listof Str) is one of:
;; ⋆ empty
;; ⋆ (cons Str (listof Str))
Template for processing a list of strings

;; listof-str-template: (listof Str) → Any
(define (listof-str-template los)
  (cond [(empty? los) . . . ]
        [(cons? los) . . . ]))

The second test can be replaced by else.

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

Now we go a step further.

Because (rest los) also is of type (listof Str), we apply the same computation to it – that is, we apply listof-str-template.

Here is the resulting template for a function consuming a (listof Str), which matches the data definition:

;; listof-str-template: (listof Str) → Any
(define (listof-str-template los)
  (cond [(empty? los) . . . ]
        [else (. . . (first los) . . . (rest los) . . . ]))

We can now fill in the dots for a specific example. We begin by copying the template and renaming.
Example: how many concerts?

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

(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)

(define (count-concerts los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-concerts (rest los)))]))

This is a recursive function (it uses recursion).

Thinking about list functions

What does the function produce in the base case?

What does the function do to the first element in a non-empty list?

How does the function combine the value produced from the first element with the value obtained by applying the function to the rest of the list?
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))))))])
⇒ (+ 1 (count-concerts (rest (cons "a" (cons "b" empty)))))
⇒ (+ 1 (cond [false 0][else (+ 1 . . .)]))
⇒ (+ 1 (cond [false 0][else (+ 1 . . .)]))
⇒ (+ 1 (cond [false 0][else (+ 1 . . .)]))
⇒ (+ 1 (cond [false 0][else (+ 1 . . .)]))
⇒ (+ 1 (cond [false 0][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 [(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

\[
\text{(count-concerts \((\text{cons "a" (cons "b" empty)})\))}\n\Rightarrow (+ 1 (\text{count-concerts (cons "b" empty)})))\n\Rightarrow (+ 1 (+ 1 (\text{count-concerts empty}))))\n\Rightarrow (+ 1 (+ 1 0))\n\Rightarrow 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: (listof Str) → Nat
;; Examples:
(check-expect (count-waterboys empty) 0)
(check-expect (count-waterboys (cons "Waterboys" empty)) 1)
(check-expect (count-waterboys (cons "DaCapo" (cons "U2" empty))) 0)
(define (count-waterboys los) . . . )

The template is a good place to start writing code. Write the template. Then, alter it according to the specific function you want to write.

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

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

The recursive function application will be (count-string s (rest los)).

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.**

---

**A template for (listof X)**

We can generalize listof-str-template, which consumed a (listof Str), into a template for (listof X):

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

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

Sometimes, each X in a (listof X) may require further processing, as in a (listof Posn). We can express this expectation by using a template for X as a helper function.

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

We assume this generic data definition and template from now on. You do not need to write it out for every list you use.
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.

Structural recursion

The list template 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.**
**Pure structural recursion**

In *pure* structural recursion, every argument in a recursive function application is either:

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

```
(define (func lst) . . . (func (rest lst)) . . .) ;; Structural
(define (func lst x) . . . (func (rest lst) x) . . .) ;; Structural
(define (func lst x) . . . (func (process lst) x) . . .) ;; NOT Structural
(define (func lst x) . . . (func (rest lst) (math-function x)) . . .) ;; NOT Structural
```

**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.

In practice we avoid overusing `length` and `member?`. Many assignment questions will prohibit them.

**Producing lists from lists**

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

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

;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons −12 empty)))
  (cons −2 (cons 12 empty)))
(define (negate-list lon)
  (cond [(empty? lon) . . .
        [else (. . . (first lon) . . . (negate-list (rest lon)) . . . )]))

negate-list completed

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

;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons −12 empty)))
  (cons −2 (cons 12 empty)))
(define (negate-list lon)
  (cond [(empty? lon) 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 makes sense only on a nonempty list — for instance, finding the maximum of a list of numbers.

**Exercise:** create a self-referential data definition for `NENumList`, a nonempty list of numbers. Develop a template for a function that consumes an `NENumList`. Finally, write a function to find the maximum of a nonempty list of numbers.

Contracts for nonempty lists

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

Instead of defining an explicit type `NENumList`, we can emphasize the requirement that the list must be nonempty by adding a `requires` section to the contract for a function consuming a general list.

For example, the function in the previous exercise would have the following purpose and contract:

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

The result is easy for a reader of the code to understand.
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 an explicit 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.

Many programs you write that process text will use a wrapper function. The wrapper is a small function that prepares the values passed in to a primary function, which does the actual work. Here is the primary function:

```scheme
;; (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**. It uses `string→list` to create a (listof Char) for `count-e/list`.

;; (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)))

---

**Wrapper functions**

Like helper functions, wrapper functions are a useful tool in many contexts. A primary function does the majority of the work, and the wrapper is a short function that modifies its arguments somehow to prepare the arguments for the primary function, allowing it to do its work.

---

**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)

;; sr-template: SR → Any
(define (sr-template sr)
  (. . . (sr-name sr) . . .
    (sr-salary sr) . . . ))
The following is a template function for a list of salary records.

;;; listof-sr-template: (listof SR) → Any
(define (listof-sr-template lst)
  (cond [(empty? lst) . . . ]
       [else (. . . (sr-template (first lst)) . . .
                   (listof-sr-template (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 (listof-sr-template lst)
  (cond [(empty? lst) . . . ]
       [else (. . . (sr-name (first lst)) . . .
                   (sr-salary (first lst)) . . .
                   (listof-sr-template (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

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.

\[
([\text{empty? lst}] \text{ empty})
\]

For the self-referential case, \((\text{first lst})\) 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 \(\text{sr} \rightarrow \text{tr}\).

In other words, to produce the answer, \((\text{sr} \rightarrow \text{tr} (\text{first lst}))\) should be \text{consed} onto \((\text{compute-taxes} (\text{rest lst}))\).

\[
;; (\text{compute-taxes lst}) \text{ produces a list of tax records, one for each salary record in lst}
;; \text{compute-taxes: (listof SR)} \rightarrow (\text{listof TR})
\]

\[
\begin{align*}
\text{(define} & \text{(compute-taxes lst) } \\
\text{ (cond} & \text{ [(empty? lst) empty]} \\
\text{ [else cons (sr \rightarrow tr (first lst))} \\
\text{ (compute-taxes (rest lst)))]})
\end{align*}
\]

The function \(\text{sr} \rightarrow \text{tr}\) uses the SR template (plus our previously-written function \text{tax-payable}).

\[
;; (\text{sr} \rightarrow \text{tr} \text{ sr}) \text{ produces a tax record for sr}
;; \text{sr} \rightarrow \text{tr: SR} \rightarrow \text{TR}
;; \text{Example:}
\]

\[
\begin{align*}
\text{(check-expect} & \text{(sr \rightarrow tr (make-sr "Jane Doe" 50000))} \\
\text{ (make-tr } & \text{"Jane Doe" 7923.29))}
\end{align*}
\]

\[
\begin{align*}
\text{(define} & \text{(sr \rightarrow tr sr)} \\
\text{ (make-tr} & \text{(sr-name sr)} \\
\text{ (tax-payable (sr-salary sr))))}
\end{align*}
\]
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))))
⇒ (compute-taxes
   (cons (make-tr "Jane Doe" 7923.29)
         (cons (make-sr "Da Kou" 15500)
               (cons (make-sr "MusaAlKhwarizmi" 100000) empty))))
⇒ (compute-taxes
   (cons (make-tr "Jane Doe" 7923.29)
         (cons (make-tr "Da Kou" 2325)
               (cons (make-sr "MusaAlKhwarizmi" 100000) empty))))
⇒ (compute-taxes
   (cons (make-tr "Jane Doe" 7923.29)
         (cons (make-tr "Da Kou" 2325)
               (cons (make-tr "MusaAlKhwarizmi" 19407.01)
                     (compute-taxes empty))))
⇒ (compute-taxes
   (cons (make-tr "Jane Doe" 7923.29)
         (cons (make-tr "Da Kou" 2325)
               (cons (make-tr "MusaAlKhwarizmi" 19407.01) empty)))
⇒ (compute-taxes
   (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.