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

- The simplest list is the empty list: empty in Racket.
- 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))))
```

$\Rightarrow$ `"a"`

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

$\Rightarrow$ `(cons "b" (cons "c" empty))`

Substitution rules:

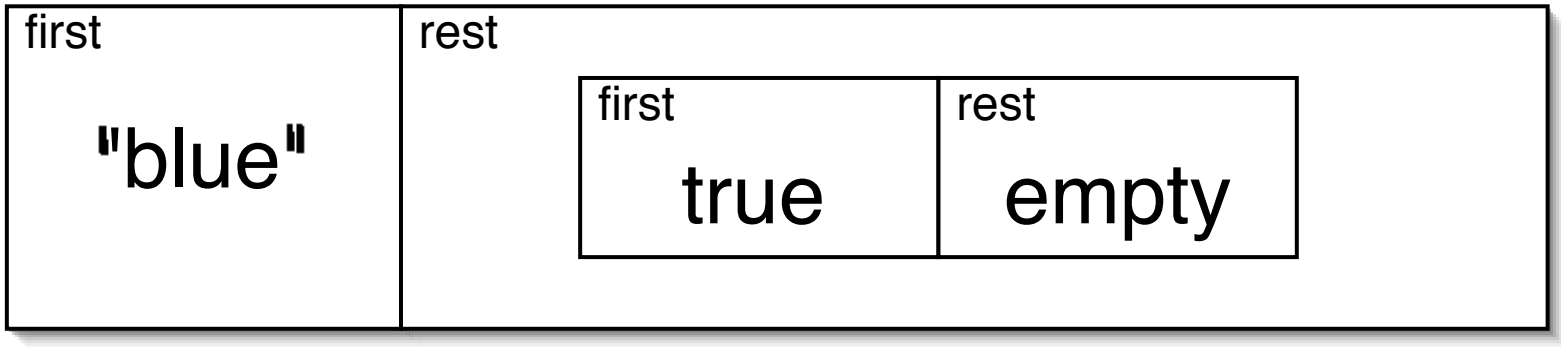
`(first (cons elt alist))` $\Rightarrow$ `elt`

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

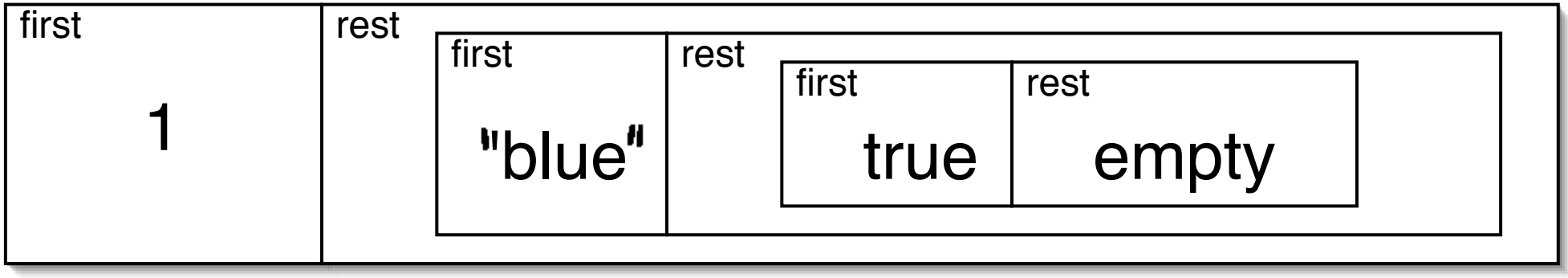
The functions consume nonempty lists only.

Nested boxes visualization

(cons "blue" (cons true empty))

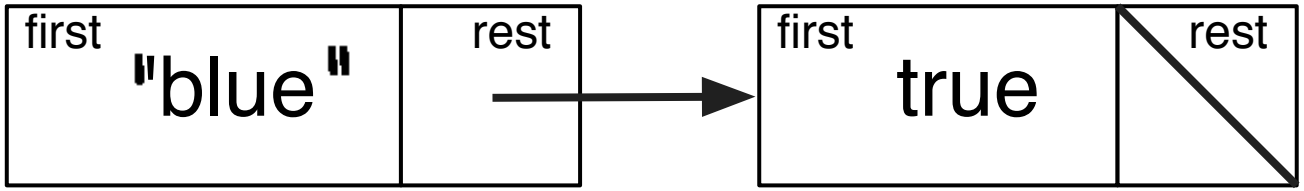


(cons 1 (cons "blue" (cons true empty)))

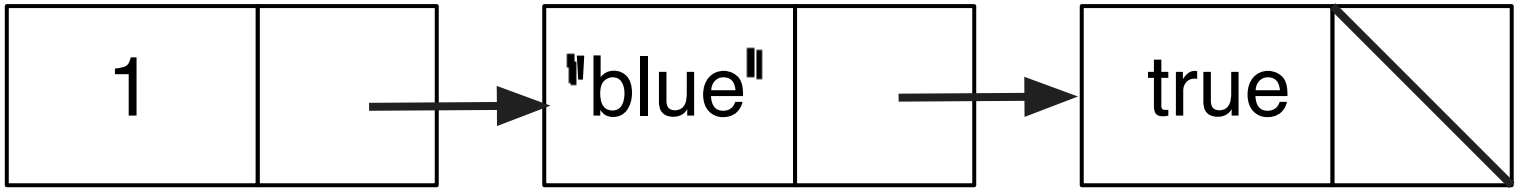


Box-and-pointer visualization

(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

In Module 03, we introduced a new type ExtNum using a data definition:

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

We used ExtNum values in functions, and included the new type name in our contracts.

```
;; 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"])
```

We noted that functions that consume an ExtNum often have the same structure (captured in a general template for the type).

We will continue this general process for all new types we introduce.

Data-directed design

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.

How do we define a list?

```
;; 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 empty
        ;; ...
```

This won't work. But, if the list is not empty, we always have an element followed by a list.

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

... this is a self-referential (recursive) data definition!

This definition is a good start, but we want to be a bit more specific about what types of values are a list.

Recursive data definition of a list of integers

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

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

This is a **recursive** definition, with a **base** case, and a **recursive** case.

Which of these are lists of integers?

empty

(cons 5 empty)

(cons −1 (cons 5 empty))

(cons −1 (cons 5 3))

Constructing a (listof Int) template

A template is a partially completed function that shows how to consume date of a given type. Let's develop a template for a (listof Int). The predicates empty? and cons? will be helpful.

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

The second test can be replaced by else.

Developing the template: use list selectors

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

Now, since rest aloi is also a list of integers, apply listof-int-template to it. This recursive application of a function solves the problem for the rest of the list. Combine the solution *in some way* with the first element in the list to solve the original problem.

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

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

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

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.
;; my-length: (listof Int) → Nat
;; Examples:
(check-expect (my-length empty) 0)
(check-expect (my-length (cons 3  (cons -2 empty))) 2)

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

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)))
```

$\Rightarrow$ (+ 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.

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

Body: Use examples to fill in the template.

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

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

The (listof Int) template revisited

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

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

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

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

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

Types used in contracts

We have now seen lots of different types of data used in our functions. All can be used in our contracts:

- Types which are built into Racket. For example, Num, Int, Str, and Bool.
- Types defined only inside a data definition, such as (listof Int) or (anyof Bool Str).

listof notation in contracts

The data definitions for (listof Int) and (listof Str) are the same, other than the element type. We will extend the definition more generally.

You do not need to include this definition on assignments.

;; A (listof τ), for a type τ, is one of:

;; * empty

;; * (cons τ (listof τ))

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

Always replace τ with the most specific type available.

General list templates

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

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

You will use this template many times!

Templates as generalizations

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

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

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

Use the templates!

Example: count-apples

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

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

Generalizing count-apples

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

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

Extra information in a list function

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

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

Built-in list functions

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

The built-in pedicate 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?, cons?, length, and member? in the code that you write.

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

Producing lists from lists

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

For example:

(negate-list empty) $\Rightarrow$ empty

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

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

Building negate-list from template

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

negate-list completed

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

Condensed trace of negate-list

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

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

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

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

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

⇒ (cons −2 (cons 12 empty))

Removing elements from a list

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

Complete singles to produce a list with only the first occurrence of each element. You may use one of the built-in functions remove-all or remove.

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

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

Wrapping a function in another function

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

Consider using a wrapper function

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

Wrapper functions will be very useful when designing many functions that consume and process strings. The "wrapped" helper function will typically do most of the work of solving the problem.

Strings and characters

Strings are made up of zero or more character values.

In Racket, characters are a separate type.

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

Built-in functions: char<? to compare two lower-case letters, two upper-case letters, or two digits.

Built-in predicates char-upper-case?.

The course web page has more information about characters.

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

Strings and lists of characters

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

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

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

We use lists of characters in labs, assignments, and exams. We will use helper functions that accept lists of characters to actually solve many string problems.

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 (using the list template) 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*).

String function example

```
;; Example of Wrapped helper function
;; listof-char-template: (listof Char) → (listof Char)
;; Note: You do not need to include examples and tests for the
;;              wrapped helper function.
(define (listof-char-template: loc) (...))

;; Example of Wrapper function
;; string-template: Str → Str
(define (string-template s)
    (list->string (listof-char-template (string->list s))))
```

Canadianizing a string

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

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

You will write canadianize-list in lab.

Determining portions of a total

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

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

For an empty list, we produce the empty list.

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

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

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

We create a function portions/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 an argument to a function application of portions/total.

Nonempty lists

Sometimes a given computation only makes sense on a nonempty list – for instance, finding the maximum of a list of numbers. If the function requires that a parameter is a nonempty list, we can provide a new data definition for non-empty lists.

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

Template for NeIn

```
;; neln-template: NeIn → Any
(define (neln-template nelst)
    (cond [(empty? (rest nelst)) (... (first nelst) ...)]
          [else (... (first nelst) ...
                     (neln-template (rest nelst)) ...)]))
```

```
;; (max-list lon) produces the maximum element of lon
;; max-list: NeIn → Num

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

Functions with multiple base cases

Suppose we wish to determine whether all values in a list of numbers are equal.

What should the function produce if the list is empty?

What if the list has only one item?

Goals of this module

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

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

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

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

You should be aware of situations that require wrapper functions.

You should understand how to use (listof . . .) notation in contracts, as well as how to add restrictions on lists and list values.