Chapter 12
Lists

Congratulations! You are no longer a beginning student. Change your language in DrRacket to `Intermediate Student with lambda`. We shall refer to this language as `ISL+`. All your programs written in `BSL` will still work in `ISL+`. In fact, `BSL` is a subset of `ISL+`.

In this chapter, we begin to study data of arbitrary size. By data of arbitrary size, we mean data whose size is not constant. In a practical sense, this means that we shall begin to learn how to solve problems involving compound data that may have a varying number of elements. This does not mean that data may be of infinite size. Every instance of our data will be of finite size. Unlike structures, however, not every instance shall be of the same size.

Data of arbitrary size is a natural part of human life. For example, it is typical for persons to make a grocery list before going to the supermarket. My grocery list, of course, may very well be different than your grocery list. This is no surprise given that we may have different tastes. For example, a vegetarian's grocery list will not contain animal proteins while a non-vegetarian will contain animal proteins. Beside content, there is another important difference between these lists. They may contain a different number of items. For example, one grocery list may contain milk, walnuts, and bread while another grocery list may contain sliced turkey, milk, bread, onions, and salmon. Yet, someone else's grocery list may be empty. Regardless of the size, we all agree that they are valid grocery lists. That is a grocery list is data of arbitrary size.

Lists may be used to represent much more than just the groceries that we need to buy. For example, in Aliens Attack we would like to have more than one alien and we would like to allow the player to shoot more than one shot at a time. Consider carefully what it means to have multiple aliens and multiple shots in Aliens Attack. Is the number of aliens constant? Is the number of shots constant? Clearly, the answer to both questions is no. As the game advances the number of aliens decreases as they are hit. That is, the number of aliens is not constant. Similarly, the number of shots varies. At the beginning of the game, the number of shots is 0. Every time the player shoots the number of shots increases. Every time a shot hits an alien or goes off the scene, the number of shots decreases. This tells us that aliens and shots are data of

M. T. Morazán, *Animated Problem Solving*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-85091-3_12

arbitrary size just as a grocery list. Thus, we may consider using a list to represent the aliens and the shots.

65 Creating and Accessing Lists in ISL+

Given that many different types of data are naturally represented as lists, ISL+ provides support for lists. The smallest possible list is the empty list. In ISL+, the empty list is represented as '(). We can use the empty list to define many different types of list. For example, an empty grocery list, an empty list of aliens, and an empty list of shots may be defined as follows:

```
(define E-GLIST '())
(define E-LOA   '())
(define E-LOS   '())

(check-expect (empty? E-GLIST) #true)
(check-expect (empty? E-LOA) #true)
(check-expect (empty? E-LOS) #true)
```

In essence, '() is an ISL+ constant that represents any type of empty list. It is the basis of all lists. The ISL+ predicate empty? returns true if its input is '() and #false otherwise.

Non-empty lists, may be constructed by adding an element to the front of an existing list. Just like constructors are needed to create structures, a constructor is needed to build a new list. In ISL+ this function is cons (for **cons**truct). The signature and purpose statement for cons are the following:

```
any list → list
Purpose: To construct a new list by adding the given value
         to the front of the given list
```

The contract is stating that the first argument may be of any type, that the second argument must be a list (of any type), and that it returns a list (of any type). This means that cons is a generic function that may be used to construct any type of list. We can now use cons to build a grocery list, a list of aliens, and a list of shots of any (finite) size. For example, a grocery list with 2 elements, a list of 5 aliens, and list of 3 shots may be defined as follows:

```
(define A-GLIST (cons "milk"
                      (cons "apples"
                            E-GLIST)))
(define A-LOA   (cons (make-posn 10 2)
                      (cons (make-posn 5 12)
                            (cons (make-posn 4 8)
                                  (cons (make-posn 15 7)
                                        (cons (make-posn 6 6)
                                              E-LOA))))))
```

```
(define A-LOS   (cons (make-posn 4 2)
                      (cons NO-SHOT
                            (cons (make-posn 17 3)
                                  E-LOS))))

(check-expect (cons? E-GLIST) #false)
(check-expect (cons? E-LOA) #false)
(check-expect (cons? E-LOS) #false)
(check-expect (cons? A-GLIST) #true)
(check-expect (cons? A-LOA) #true)
(check-expect (cons? A-LOS) #true)
```

The arguments to `cons` are lined up for you to easily confirm that its second argument is always a list. That is, the second argument is always a list constructed using `cons` or is the empty list (i.e., `E-GLIST`, `E-LOA`, or `E-LOS`). The first argument varies depending on the type of list constructed. For the grocery list, the first argument is always a string. For a list of aliens, the first argument is always an alien. For the list of shots, the first argument is always a shot. Now, it should be even clearer that lists, just as structures, are generic. The tests use the ISL+ predicate `cons?` that returns true when its input is a non-empty list. Otherwise, it returns false.

As with structures, selectors are needed to extract the elements used to construct a list. Given that a non-empty list is always constructed using `cons`, two selectors are needed to extract the components of a list. The selector to extract the first element of a given list is `first`. The selector to extract the list containing the rest of the elements (not including the first element) of a given list is `rest`. Care must be taken when using these selector functions because the empty list does not have a first element nor does it have a value for the rest of the elements. If either list-selector function is applied to the empty list, an error is thrown. The following tests illustrate how the list-selector functions work:

```
(check-error (first E-GLIST))
(check-error (first E-LOA))
(check-error (first E-LOS))

(check-expect (first A-GLIST) "milk")
(check-expect (first A-LOA)   (make-posn 10 2))
(check-expect (first A-LOS)   (make-posn 4 2))

(check-error (rest E-GLIST))
(check-error (rest E-LOA))
(check-error (rest E-LOS))

(check-expect (rest A-GLIST) (cons "apples" E-GLIST))
```

```
(check-expect (rest A-LOA)
              (cons (make-posn 5 12)
                    (cons (make-posn 4 8)
                          (cons (make-posn 15 7)
                                (cons (make-posn 6 6)
                                      E-LOA)))))
(check-expect (rest A-LOS)
              (cons NO-SHOT (cons (make-posn 17 3) E-LOS)))
```

Here a special form of `check-error` is used that only checks if an error is thrown. It does not require an error message to check. This form of `check-error` is particularly useful when testing functions whose error messages we did not design.

How is the second element of a list extracted? The third element? To achieve extractions beyond the first list element you need to compose calls to `first` and `rest`. For example, the second element of `A-LOS`, `NO-SHOT`, is given by `(first (rest A-LOS))`. Let us examine how this works:

```
(first (rest A-LOA)) = (first
                        (rest (cons (make-posn 4 2)
                                    (cons NO-SHOT
                                          (cons
                                            (make-posn 17 3)
                                            E-LOS)))
                      = (first (cons NO-SHOT
                                     (cons (make-posn 17 3)
                                           E-LOS)))
                      = NO-SHOT
```

The first step substitutes `A-LOA` for its value. In the second step, the inner most function application (using `rest`) is evaluated and is replaced with its value. The third step applies the outermost function application to obtain `NO-SHOT`.

* **Ex. 113 —** Design and implement a function, `third-of-list`, to extract the third element of a list. Use compositions of `first` and `rest`.

* **Ex. 114 —** Design and implement a function, `fourth-of-list`, to extract the fourth element of a list. Use compositions of `first` and `rest`.

* **Ex. 115 —** Design and implement a function, `fifth-of-list`, to extract the fifth element of a list. Use compositions of `first` and `rest`.

** **Ex. 116 —** Design and implement a function, `tenth-of-list`, to extract the tenth element of a list. Use compositions of `first` and `rest`.

Extracting the second, third, fourth, and so on up to the eighth element of a list is common enough that `ISL+` provides functions to do so directly. Not surprisingly, these functions are called `second`, `third`, `fourth`, and so on up to `eighth`. Be mindful when using these functions because if given a list that is too short they throw an error. These function provide a useful shorthand notation to access list elements up to the eighth. From the ninth on you must write your own function.

66 Shorthand for Building Lists

Consider constructing a list of first 7 digits. Your code may look something like this:

```
(define SEVEN-DIGITS
        (cons
          0
          (cons 1
                (cons 2
                      (cons 3
                            (cons 4
                                  (cons 5
                                        (cons 6 empty))))))))
(check-expect (first SEVEN-DIGITS)   0)
(check-expect (second SEVEN-DIGITS)  1)
(check-expect (third SEVEN-DIGITS)   2)
(check-expect (fourth SEVEN-DIGITS)  3)
(check-expect (fifth SEVEN-DIGITS)   4)
(check-expect (sixth SEVEN-DIGITS)   5)
(check-expect (seventh SEVEN-DIGITS) 6)
```

Observe that there is a lot of repetition in the code to construct the list. Specifically, there is an application of `cons` for every element of the list. This may not be too cumbersome for a list with seven elements, but what if you are now asked to define a list with the integers in `[0..19]`? You would need to write `cons` twenty times. To avoid all this repetition `ISL+` provides three shorthand constructors for lists.

The first is used when the elements of the list are known in advance. A *quoted list* has the elements listed inside parenthesis preceded by a `'`. For example, we may refactor the `SEVEN-DIGITS` definition as follows:

```
(define SEVEN-DIGITS '(0 1 2 3 4 5 6))

(check-expect (first   SEVEN-DIGITS)  0)
(check-expect (second  SEVEN-DIGITS)  1)
(check-expect (third   SEVEN-DIGITS)  2)
(check-expect (fourth  SEVEN-DIGITS)  3)
(check-expect (fifth   SEVEN-DIGITS)  4)
(check-expect (sixth   SEVEN-DIGITS)  5)
(check-expect (seventh SEVEN-DIGITS)  6)
```

Using a quoted list eliminates the need to repeatedly write `cons` to construct the list. It is important to note that nothing after the quote (inside the parenthesis) is evaluated. That means all the listed elements are literal values in the list. This is important because there can be no expressions that need to be evaluated inside the parenthesis. For example, consider the following code:

```
(define SOME-SQRS1 (cons (sqr 0)
                         (cons (sqr 1)
                               (cons (sqr 2)
                                     empty))))

(define SOME-SQRS2 '((sqr 0) (sqr 1) (sqr 2)))

(check-expect SOME-SQRS1 '(0 1 4))
(check-expect (not (equal? SOME-SQRS1 SOME-SQRS2)) #true)
```

Both tests pass. For `SOME-SQRS1` the expressions using `sqr` are evaluated. For `SOME-SQRS2` they are not evaluated because they are inside a quoted list. Instead of being evaluated, they are considered literal sublists. Thus, the value of `SOME-SQRS2` is:

```
(cons (cons 'sqr (cons 0 empty))
      (cons (cons 'sqr (cons 1 empty))
            (cons (cons 'sqr (cons 2 empty))
                  empty)))
```

Naturally, you must be asking yourself if there is shorthand notation to create a list from evaluated expressions. After all, it would be nice not to repeatedly write `cons` to define `SOME-SQRS1`. The shorthand provided by `ISL+` is `list`. This is a list-constructor function that evaluates all of its arguments and creates a list of the results. The number of arguments is arbitrary. If no arguments are provided `list` returns `'()`. We can refactor the `SOME-SQRS2` definition and update the tests as follows:

```
(define SOME-SQRS1 (cons (sqr 0)
                         (cons (sqr 1)
                               (cons (sqr 2)
                                     empty))))

(define SOME-SQRS2 (list (sqr 0) (sqr 1) (sqr 2)))

(check-expect SOME-SQRS1 '(0 1 4))
(check-expect (equal? SOME-SQRS1 SOME-SQRS2) #true)
```

Use `list` when you want all the expressions to be evaluated to construct a list.

The third shorthand is a *quasiquoted list*. A quasiquoted list is a combination of `'` and `list`. Instead of preceding the opening parentheses with a `'`, it is preceded with `` ` ``. The `` ` `` indicates that some subexpression inside the parenthesis may have to be evaluated. To indicate that an expression needs to be evaluated, it must be preceded by a `,`. If there are no expressions preceded by a comma inside the parenthesis, then `` ` `` and `'` yield the same value when evaluated. To illustrate the use of quasiquote consider the following code:

```
(define X 2)
(define Y 3)
(define Z 4)
```

```
(define A-LIST  '((add1 X)  J  (* Z Y)))
(define A-LIST2 `(,(add1 X) J ,(* Z Y)))

(check-expect (first A-LIST)  '(add1 X))
(check-expect (first A-LIST2) 3)

(check-expect (second A-LIST)  'J)
(check-expect (second A-LIST2) 'J)

(check-expect (third A-LIST)  '(* Z Y))
(check-expect (third A-LIST2) 12)
```

The expressions `(add1 X)` and `(* Z Y)` are evaluated to construct `A-LIST2` because they are preceded by a comma in a quasiquoted list. The same expressions are treated as literal values for `A-LIST` because they are inside a quoted list.

67 Recursive Data Definitions

We have explored how to create and access lists. As part of our exploration, we have discovered that lists may be used to store different types of data. We have not yet explored how to process a list. To do so we first need to be able to develop a data definition and a function template for a given type of list. Consider defining a list of numbers. Let us look at some examples based on our knowledge of lists so far:

```
'()
(cons 87 '())
(cons 24 (cons 87 '()))
(cons 16 (cons 24 (cons 87 '())))
(cons 31 (cons 16 (cons 24 (cons 87 '()))))
```

If you think about it carefully, you can see a distinct pattern. There are two list-of-numbers varieties. The first is the empty list of numbers. The second is a non-empty list of numbers constructed using `cons`. This immediately suggests that to define a list of numbers, we need a data definition with two varieties:

```
;; A list of numbers (lon) is either
;;  1. '()
;;  2. (cons ??? ???)
```

Now, we must be precise about the types of the arguments to `cons`. Given that we are defining a list of numbers, it is clear that the first argument must be a number. This takes us a step closer to the needed data definition:

```
;; A list of numbers (lon) is either
;;  1. '()
;;  2. (cons number ???)
```

What is the type of the second argument to `cons`? To answer this question, let us analyze one of our examples: `'(cons 31 (cons 16 (cons 24 (cons 87 '())))`. In this example, `31` is the first number of the list. Clearly, it fulfills what is required by the data definition we have so far. What is `'(cons 16 (cons 24 (cons 87 '()))`? If you think about it carefully, the inescapable conclusion is that it is a `lon`. Is this really the case? Does this make sense? Let us look at another example: `'(cons 87 '())`. Here the argument types to `cons` are number (the 87) and, once again, a `lon` (`'()`). You can verify that all other examples satisfy this pattern. This analysis is telling us that the type of the second argument to `cons` is `lon`. Thus, the `lon` data definition is:

```
;; A list of number (lon) is either
;;  1. '()
;;  2. (cons number lon)
```

A list of numbers is defined in terms of itself. That is, it is a circular data definition. A type defined in terms of itself is a *recursive* data type.

Does this make any sense? Can this data definition be used to build any list of numbers? It turns out that you can build any list of numbers using this data definition. To build the empty list of numbers, we simply use the first rule and write: `'()`. How about a list that is not empty? Here is how you build `'(cons 24 (cons 87 '()))`:

```
lon → (cons 24 lon)          using rule 2 to substitute lon
    → (cons 24
            (cons 87 lon))   using rule 2 to substitute lon
    → (cons 24
            (cons 87 empty)) using rule 1 to substitute lon
```

This is an example of a `derivation`. A derivation is the series of steps used to show how to build an instance of a data type using the varieties of a data definition. It is not difficult to see that any valid list of numbers can be derived using the above data definition. Can anything that is not a list of numbers be derived? Let us try to derive `'(cons 42 (cons -6 (cons "Hi" (cons 87 '()))))`:

```
lon → (cons 42 lon)  using rule 2 to substitute lon
    → (cons 42
             (cons -6 lon))  using rule 2 to substitute lon
```

The derivation fails because there is no `lon` variety whose first element is a `string`. Therefore, `'(cons 31 (cons 16 (cons "Hi" (cons 87 '()))))` is not a `lon`. It is also not difficult to see that according to our data definition all the elements of a `lon` must be numbers. If any element of a given list is not a number, then it is not a `lon`.

It is very likely that you have been taught to steer away from recursive data definitions. This is unfortunate because as we have seen recursive data definitions

are useful to define data of arbitrary size like lists. In fact, any data of arbitrary size requires a recursive data definition. So, why are students steered away from recursive data definitions? This is likely rooted in the fact that recursive data definitions may be nonsense. For example, someone may attempt to define a list of numbers as follows:

```
; A list of numbers (lon2) is a (cons number lon2)
```

Is this a useful data definition? Let us try to derive `'(cons 24 (cons 87 '()))`:

```
lon2 → (cons 24 lon2)          substitute lon2 using rule 2
     → (cons 24
             (cons 87 lon2))   substitute lon2 using rule 2
```

The derivation fails because we are unable to instantiate the `lon2` in `(cons 87 lon2)`. You may say that clearly it should be `'()`. In fact, this would be wrong. Nowhere in this data definition does it say that `'()` is a `lon2`. It is important to be precise with our data definitions. Otherwise, we will be unable to solve problems.

This leads to asking ourselves what constitutes a useful recursive data definition. In order to be useful, a recursive data definition must have the following characteristics:

1. At least two subtypes (varieties)
2. At least one subtype that does not contain a selfreference
3. At least one subtype that contains a selfreference

The subtypes that do not contain a selfreference are known as `base subtypes`. These are concrete values that are known to be instances of the data type defined. These concrete values break the circularity in a derivation. The subtypes that do contain a selfreference are known as `recursive subtypes`. These are the varieties that introduce circularity which endows us with the power to define data of arbitrary size. In the recursive data definition for `lon`, we have `'()` as a base subtype and `(cons number lon)` as a recursive subtype. The recursive data definition for `lon2` is not useful because it does not have a base subtype.

With our newly acquired knowledge, we can now define a representation for multiple aliens and multiple shots in Aliens Attack. Given that both are data of arbitrary size, we need a recursive data definition for each. At this point, the only recursive data type we know is list. Thus, let us try to use a list to define an arbitrary number of aliens. Remember that it must have the three characteristic above required for a useful recursive data definition. A list of aliens may be defined as follows:

```
;; A list of alien (loa) is either:
;;  1. '()
;;  2. (cons alien loa)

;; Sample instances of loa
(define E-LOA '())
(define INIT-LOA
        (list
          (make-posn 8 0)  (make-posn 9 0)  (make-posn 10 0)
          (make-posn 11 0) (make-posn 12 0) (make-posn 13 0)
```

```
(make-posn 8 1)  (make-posn 9 1)  (make-posn 10 1)
(make-posn 11 1) (make-posn 12 1) (make-posn 13 1)
(make-posn 8 2)  (make-posn 9 2)  (make-posn 10 2)
(make-posn 11 2) (make-posn 12 2) (make-posn 13 2)))
```

Observe that the `loa` data definition has two subtypes: one base subtype and one recursive subtype. In addition, following the steps of the design recipe, an instance of each variety of `loa` is defined. The initial list of aliens, `INIT-LOA`, contains 18 aliens. This ought to make the game more interesting. You are encouraged to make the initial list of aliens shorter or longer to personalize the game to your liking.

A list of shots is defined as follows:

```
;; A list of shot (los) is either:
;;  1. '()
;;  2. (cons shot los)

;; Sample instances of los
(define INIT-LOS '())
(define LOS2 (list (make-posn 8 0) (make-posn 10 5)))
```

Observe that the requirements for a useful data definition are satisfied and that sample instances are also defined.

* **Ex. 117 —** Define sample instances for `lon`.

* **Ex. 118 —** Create a data definition and sample instances for a grocery list.

* **Ex. 119 —** In Chap. 7 a data definition for a 2D-point is developed. Why or why not is this a useful recursive data definition?

*** **Ex. 120 —** Create a data definition for a composed image. A composed image may contain one or more rectangles and circles that are above, next, or overlaid in relation to each other.

68 Generic Data Definitions

The next natural step is to develop a function template for `lon`, `loa`, and `los`. However, notice that the three data definitions are almost identical. In other words, there is a lot of repetition among them. When we have repetition among expressions, an abstraction step introduces a variable to obtain a function. Can we apply an abstraction step to avoid repetitions among data definitions?

The situation we face is a bit different from abstraction over expressions. When you abstract over expressions, a difference is always a value and a variable is used to represent the value. This felt quite natural because you are familiar with variables from your Mathematics courses. The difference among the data definitions for `lon`, `loa`, and `los` is a type. For `lon` the type used is `number`. For `loa` the type used

is alien. For los the type used is shot. When abstracting over data definitions the variables used to capture the differences are type variables. Type variables represent types not instances of types (i.e., values). A type variable is needed for each difference.

Let X be the single difference among lon, loa, and los. Instead of using a concrete type (like number, alien, or shot), we write a data definition using X:

```
;; A list of X ((listof X)) is either:
;;  1. '()
;;  2. (cons X (listof X))
```

The data definition is recursive just like the data definitions for lon, loa, and los. Observe that if we plug in number for X we obtain the data definition for lon. Similarly, plugging in alien and shot for X yields, respectively, the data definitions for loa and los. The notation (listof X) is used to emphasize that a type must be plugged in to obtain a concrete data definition. Clearly, this data definition works for many different types. A data definition that works for many different types is called a generic (or parameterized) data definition.

As we shall see, a generic data definition may be used in two ways in our signatures. We may substitute X with a concrete type to specify a concrete data definition or we may use (listof X) directly. We shall start with examples of the former. The use of the latter will follow after we learn how to abstract over functions.

69 Function Templates for Lists

We can write a function template for a (listof X) and use it to define the templates for lon, loa, and los. Before that, however, we must learn to write a function template for a recursive data definition. Using the knowledge you have accumulated so far, we can begin to develop the function template for (listof X):

```
;; Sample instances of (listof X)
(define LOX1 ...)
(define LOX2 ...)

;; (listof X) ... → ...
;; Purpose: ...
(define (f-on-loX a-loX ...)
  (if (empty? a-loX)
      ...
      ...))

;; Sample expressions for f-on-loX
(define LOX1-VAL ...)
(define LOX2-VAL ...)
   ...
```

```
;; Tests using sample computations for f-on-loX
(check-expect (f-on-loX LOX1 ...) LOX1-VAL)
(check-expect (f-on-loX LOX2 ...) LOX2-VAL)
   ...

;; Tests using sample values for f-on-los
(check-expect (f-on-loX ...) ...)
   ...
```

A function template for a generic type contains all the same elements as a concrete function template: sample instances templates, a function definition template, sample expression templates, and test templates. The definition template has a conditional expression because there is variety in the data. So far, there is nothing really new here except not knowing what `X` is at this time. That is fine because when a type is plugged in for `X`, the yield is a concrete function template. The question we must answer is how do we deal with selfreferences in the data definition. Specifically, how is ... substituted in the definition template's else clause. This clause tells us how to process a list constructed using `cons`. Let us examine carefully what needs to be processed:

```
(cons X (listof X))
```

There are two different types that need to be processed: `X` and `(listof X)`. Processing an `X` is done by calling a function on an `X`. How do we process a `(listof X)`? The natural answer is to call a function that processes a `(listof X)`. In other words, the function must call itself. Let us finalize the template using this new insight:

```
;; Sample instances of (listof X)
(define LOX1 ...)
(define LOX2 ...)
...

;; (listof X) ... → ...
;; Purpose: ...
(define (f-on-loX a-loX ...)
  (if (empty? a-loX)
      ...
      ...(f-on-X (first a-loX))...
      ...(f-on-loX (rest a-loX) ...)...))

;; Sample expressions for f-on-loX
(define LOX1-VAL ...)
(define LOX2-VAL ...)
   ...
```

```
;; Tests using sample computations for f-on-loX
(check-expect (f-on-loX LOX1 ...) LOX1-VAL)
(check-expect (f-on-loX LOX2 ...) LOX2-VAL)
   ...

;; Tests using sample values for f-on-loX
(check-expect (f-on-loX ...) ...)
   ...
```

A new principle now becomes clear. Every selfreference in a data definition becomes a selfreference in the definition template and, eventually, a selfreference in a function definition. That is, *data of arbitrary size is processed by a recursive function*. Recursion that is based on the structure of your data is called *structural recursion* (or natural recursion). As before with structures, the structure of your data suggests the structure of your functions. Observe that a divide and conquer strategy naturally arises from the structure of a `(listof X)`. To solve a problem for a `(listof X)` a subproblem for an `X` and a subproblem for a (smaller) `(listof X)` must be solved. The fascinating part is that the subproblem for the smaller `(listof X)` is solved the same way as the larger `(listof X)`. When using structural recursion, it is always the case that the subproblem is smaller and, therefore, closer to a base case. This means that when structural recursion is correctly used, your code is guaranteed to eventually stop. It becomes impossible for a function to implement an infinite recursion.

70 Designing List-Processing Functions

We can use the generic data definition for a `(listof X)` to define a list of numbers:

```
A lon is a (listof number)
```

Let us take a close look at what this means. Plugging in `number` for `X` in the generic data definition for `(listof X)` yields:

```
;; A list of number (lon) is either
;;  1. '()
;;  2. (cons number lon)
```

This is exactly the data definition derived above. Plugging in `number` for `X` in the function template for a function on a `(listof X)` yields a concrete function template for a `lon`:

```
;; Sample instances of lon
(define LON1 ...)
(define LON2 ...)
...
```

```
;; lon ... → ...
;; Purpose: ...
(define (f-on-lon a-lon ...)
  (if (empty? a-lon)
      ...
      ...(f-on-number (first a-lon))...
      ...(f-on-lon (rest a-lon) ...)...))

;; Sample expressions for f-on-lon
(define LON1-VAL ...)
(define LON2-VAL ...)
   ...

;; Tests using sample computations for f-on-lon
(check-expect (f-on-lon LON1 ...) LON1-VAL)
(check-expect (f-on-lon LON2 ...) LON2-VAL)
   ...

;; Tests using sample values for f-on-lon
(check-expect (f-on-lon ...) ...)
   ...
```

We can now use the template to write functions to process a list of numbers. For example, write a function that squares a list of numbers. The design idea is to traverse the given list and construct a new list with the square of each number in the list. Specializing the template yields:

```
;; Sample instances of lon
(define LON1 '())
(define LON2 '(1 2 3 4))

;; lon → lon
;; Purpose: Return a list of the squares of the given lon
(define (square-lon a-lon)
  (if (empty? a-lon)
      '()
      (cons (sqr (first a-lon))
            (square-lon (rest a-lon)))))

;; Sample expressions for square-lon
(define LON1-VAL '())
(define LON2-VAL (cons (sqr (first '(1 2 3 4)))
                       (square-lon (rest '(1 2 3 4)))))

;; Tests using sample computations for square-lon
(check-expect (square-lon LON1) LON1-VAL)
(check-expect (square-lon LON2) LON2-VAL)
```

```
;; Tests using sample values for square-lon
(check-expect (square-lon '(10 8 4)) '(100 64 16))
```

It is worth carefully analyzing the expressions used in the function definition:

`(sqr (first a-lon))`	square of the first number in `a-lon`
`(square-lon (rest a-lon))`	list of squares obtained from the rest of `a-lon`

In the function definition `cons` is used to construct the new list. A new square is added to the front of the list of squares obtained from the rest of the given list. There is a sample expression and corresponding test for each `lon` variety. Finally, as always, there is one or more tests using sample values.

To close this chapter, it is worth noting that there is a popular legend among students and professionals that recursion is hard. If you understand the design of `square-lon`, then you know better. Recursion is not hard nor should it be feared. Understanding how to design solutions to problems based on data definitions makes recursion quite natural. In the next chapter, you will gain more experience designing recursive functions. Enjoy the journey!

* **Ex. 121 —** Design a function to cube a given `lon`.

*** **Ex. 122 —** Design a function that returns a list of the lengths of all the strings in a list of strings. Make sure you have a data definition for all the list data types needed.

**** **Ex. 123 —** A student's backpack may be modeled using a `(listof symbol)`. For example, a backpack containing a notebook, a laptop, and a pen is represented as '(notebook laptop pen). Design a function to determine if a given item is contained in a given backpack.

*** **Ex. 124 —** For the previous exercise, do you need a conditional to define the predicate as suggested by the data definition for `(listof X)`? If you used a conditional for the previous exercise refactor your solution to not use a conditional.

71 What Have We Learned in This Chapter?

The important lessons of Chap. 12 are summarized as follows:

- Data of arbitrary size occurs naturally in real life and in virtual worlds.
- A list is data of arbitrary size with two varieties:
  - `'()`
  - A list built using `cons`

- The constructor for a non-empty list, `cons`, builds a new list by adding an element to the front of an existing list.
- The selectors for lists are `first` and `rest`.
- When using an error-throwing function, we did not write use `check-error` to test that an error is thrown.
- A quoted list may be used when all the list values are known and can be enumerated.
- The function `list` may be used when the expressions for each list value are known and can be enumerated.
- A quasiquoted list may be used when for the list elements either a value or an expression for the value is known. Expressions that need to be evaluated must be preceded by a comma.
- A recursive data definition is needed to define data of arbitrary size like lists.
- A recursive data definition always has a variety of subtypes of which at least one is a base subtype and at least one is a recursive subtype.
- A recursive subtype contains a selfreference to its supertype.
- A generic data definition is an abstraction over types and may be used to define many different concrete data types.
- For `(listof X)` there are two varieties: `'()` is the base subtype and `(cons X (listof X))` is the recursive subtype.
- A generic data definition leads to a generic function template.
- In a function template for data of arbitrary size, the definition template has a recursive call for every selfreference in the data it is designed to process.
- Data of arbitrary size is processed using a recursive function.
- Recursion based on the structure of the data is called structural recursion.
- Structural recursion implements a divide and conquer design.
- Proper use of structural recursion guarantees that functions are not infinite recursions.
- A concrete type and a concrete function template is obtained from a generic type and its generic template by plugging in a type for each type variable.
- A concrete type and its concrete template are used to design functions. For example, `(listof number)` and its function template are used to design any function that processes a list of numbers.