Using the List Template

CS 5010 Program Design Paradigms
Lesson 4.2

Learning Objectives

At the end of this lesson you should be able to:

- Write down the template for list data.

- Use the template for list data to write simple functions on lists.

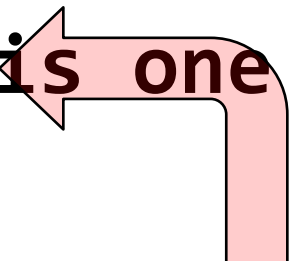# Review: The ListOfX Data Definition

```
A ListOfX is one of
-- empty
-- (cons X ListOfX)
```

Here is the data definition for a list of X's

This definition is self-referential.

```
A ListOfX is one of
-- empty
-- (cons X ListOfX)
```

Template for List data
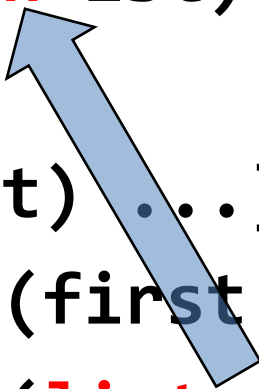
```
;; list-fn : ListOfX -> ??
(define (list-fn lst)
  (cond
     [(empty? lst) ...]
     [else (... (first lst)
                (list-fn (rest lst)))]))
```

Here is the template for list data. It is just like a template for mixed data, with one change. In the second case, we get to use not just **(rest lst)** but **(list-fn (rest lst))** . This important change is shown in red.

Observe that **lst** is non-empty when **first** and **rest** are called, so their WHERE-clauses are satisfied.

This template is *self-referential*

```
;; list-fn : ListOfX -> ??
(define (list-fn lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
            (list-fn (rest lst)))]))
```

(rest lst) is a **ListOfX**, so call **list-fn** on it

New Slogan: Self-reference in the data definition leads to self-reference in the template.

Let's add this to the recipe for writing a template

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs as many cond clauses as subclasses that the data definition distinguishes.
How do the subclasses differ from each other?	Use the differences to formulate a condition per clause.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the clause.
Does the data definition use self-references?	Formulate ``natural recursions'' for the template to represent the self-references of the data definition.

We got the list template by following the template recipe and adding one more step.

Let's see how the four steps in the template recipe show up in the list template.

```
;; list-fn : ListOfX -> ??
(define (list-fn lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (list-fn (rest lst))]))
```

1. Write a cond clause with the correct number of clauses.
2. Write predicates that distinguish the cases.
3. For mixed data, add selectors
4. For recursive data, add a recursive call

Observe that **(cons X ListOfX)** was a structured value, and that **(first lst)** and **(rest lst)** were the appropriate selector expressions

From Template to Function Definition

- Remember that when we use a template, all we do is fill in the blanks.

- For each blank, we had a question to answer:
  - "What's the answer for a red light?"
  - "What's the answer for a yellow light?"
  - "What's the answer for a green light?"

- The questions are the same, no matter what the function is.

Template Questions for TLState

```
;; tls-fn : TLState -> ??
;(define (tls-fn tls)
;   (cond
;      [(string=? tls "red") ...]
;      [(string=? tls "yellow") ...]
;      [(string=? tls "green") ...]))
```

What's the answer for "red"?

What's the answer for "yellow"?

What's the answer for "green"?

The questions are the same, no matter what function we are defining.

To finish the function definition, all we do is to fill in the blanks with the answers.

Template questions for ListOfX

```
;; list-fn : ListOfX -> ??
(define (list-fn lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
              (list-fn (rest lst)))]))
```

What's the answer for the empty list?

Here are the template questions for the list template.

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Let's do some examples

- We'll be working with the list template a lot, so let's do some examples to illustrate how it goes.
- We'll do 5 examples, starting with one that's very simple and working up to more complicated ones.

Example 1: lon-length

```
lon-length : ListOfNumber -> Number
GIVEN: a ListOfNumber
RETURNS: its length
EXAMPLES:
(lon-length empty) = 0
(lon-length (cons 11 empty)) = 1
(lon-length (cons 33 (cons 11 empty))) = 2
STRATEGY: Use template for ListOfNumber on lst
```

Example 1: lon-length

```
lon-length : LON -> Number
Given a LON, find its length
(define (lon-length lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (lon-length (rest lst)))]))
```

We start by copying the template and changing list-fn to lon-length.

Example 1: lon-length

```
lon-length : LON -> Number
Given a LON, find its length
(define (lon-length lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
              (lon-length (rest lst)))]))
```

What's the answer for the empty list?

Next, we answer the template questions.

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 1: lon-length

```
lon-length : LON -> Number
Given a LON, find its length
(define (lon-length lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (first lst)
             (lon-length (rest lst)))]))
```
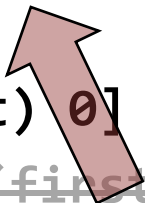
What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Next, we answer the template questions.

The code is self-referential, too

```
lon-length : LON -> Number
Given a LON, find its length
(define (lon-length lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (first lst)
              (lon-length (rest lst)))]))
```

Self-reference in the data definition leads to self-reference in the template;
Self-reference in the template leads to self-reference in the code.

Example 2: lon-sum

```
lon-sum : LON -> Number
GIVEN: a list of numbers
RETURNS: the sum of the numbers in the list
EXAMPLES:
(lon-sum empty) = 0
(lon-sum (cons 11 empty)) = 11
(lon-sum (cons 33 (cons 11 empty))) = 44
(lon-sum (cons 10 (cons 20 (cons 3 empty)))) = 33
STRATEGY: Use template for LON on lst
```

Here's another example

Example 2: lon-sum

```
lon-sum : LON -> Number
(define (lon-sum lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (lon-sum (rest lst)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 2: lon-sum

```
lon-sum : LON -> Number
(define (lon-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+   (first lst)
               (lon-sum (rest lst)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Watch this work:

```
(lon-sum (cons 11 (cons 22 (cons 33 empty)))))
= (+ 11  (lon-sum (cons 22 (cons 33 empty)))))
= (+ 11  (+ 22    (lon-sum (cons 33 empty)))))
= (+ 11  (+ 22    (+ 33    (lon-sum empty)))))
= (+ 11  (+ 22    (+ 33    0)))
= (+ 11  (+ 22    33))
= (+ 11  55)
= 66
```

Example 3: double-all

```
double-all : LON -> LON
GIVEN: a LON,
RETURNS: a list just like the original, but
         with each number doubled
EXAMPLES:
(double-all empty) = empty
(double-all (cons 11 empty))
         = (cons 22 empty)
(double-all (cons 33 (cons 11 empty)))
         = (cons 66 (cons 22 empty))
STRATEGY: Use template for LON on lst
```

Example 3: double-all

```
double-all : LON -> LON
(define (double-all lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
              (double-all (rest lst)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 3: double-all

```
double-all : LON -> LON
(define (double-all lst)
  (cond
    [(empty? lst) empty ]
    [else (cons (* 2 (first lst))
                (double-all (rest lst)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 4: remove-evens

- For this one, we'll need to specialize to integers.

```
A ListOfIntegers (LOI) is one of
-- empty
-- (cons Integer LOI)
```

Example 4: remove-evens

```
remove-evens : LOI -> LOI
GIVEN: a LOI,
RETURNS: a list just like the original, but with all the
  even numbers removed
EXAMPLES:
(remove-evens empty) = empty
(remove-evens (cons 12 empty)) = empty
(define list-22-11-13-46-7
  (cons 22 (cons 11 (cons 13 (cons 46 (cons 7 empty))))))
(remove-evens list-22-11-13-46-7)
  = (cons 11 (cons 13 (cons 7 empty)))
STRATEGY: Use template for LOI on lst
```

Example 4: remove-evens

```
remove-evens : LOI -> LOI
(define (remove-evens lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
              (remove-evens (rest lst)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 4: remove-evens

```
remove-evens : LOI -> LOI
(define (remove-evens lst)
  (cond
    [(empty? lst) empty]
    [else (if (even? (first lst))
              (remove-evens (rest lst))
              (cons (first lst)
                    (remove-evens (rest lst))))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 4: remove-evens

```
remove-evens : LOI -> LOI
(define (remove-evens lst)
  (cond
    [(empty? lst) empty]
    [else (if (even? (first lst))
              (remove-evens (rest lst))
              (cons (first lst)
                    (remove-evens (rest lst))))]))
```

Observe that this is a legal functional combination of **(first lst)** and **(remove-evens (rest lst))** .

Example 4: remove-evens

```
remove-evens : LOI -> LOI
(define (remove-evens lst)
  (cond
    [(empty? lst) empty]
    [(even? (first lst))
     (remove-evens (rest lst))]
    [else (cons (first lst)
                (remove-evens (rest lst)))]))
```

This version is OK, too. The template is just a way for you to get started writing your function definition. It's OK to vary it a little if it leads to more readable code.

Example 5: remove-first-even

```
remove-first-even : LOI -> LOI
GIVEN: a LOI,
RETURNS: a list just like the original, but with all the
    even numbers removed
EXAMPLES:
(remove-first-even empty) = empty
(remove-first-even (cons 12 empty)) = empty
(define list-22-11-13-46-7
    (cons 22 (cons 11 (cons 13 (cons 46 (cons 7 empty))))))
(remove-first-even list-22-11-13-46-7)
    = (cons 11 (cons 13 (cons (cons 46 (cons 7 empty)))))
STRATEGY: Use template for LOI on lst
```

Why is this not a good set of examples?

Answer: None of them show what happens when the first element of the list is odd

Example 5: remove-first-even

```
remove-first-even : LOI -> LOI
(define (remove-first-even lst
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (remove-first-even (rest lst)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 5: remove-first-even

```
remove-first-even : LOI -> LOI
(define (remove-first-even lst)
  (cond
    [(empty? lst) empty]
    [else (... (first lst)
              (remove-first-even (rest lst)))]))
```

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 5: remove-first-even

```
remove-first-even : LOI -> LOI
(define (remove-first-even lst)
  (cond
    [(empty? lst) empty]
    [else (if (even? (first lst))
             (rest lst)
             (cons (first lst)
                   (remove-first-even (rest lst)))])))
```

This is OK: you don't have to recur if you don't need to.

Example 5: remove-first-even

```
remove-first-even : LOI -> LOI
(define (remove-first-even lst)
  (cond
    [(empty? lst) empty]
    [(even? (first lst))
     (rest lst)]
    [(cons (first lst)
           (remove-first-even (rest lst)))]))
```

Again, here's another version of remove-first-even that is acceptable. It's OK to vary the template, but you'll be less likely to make mistakes if you stick close to the template.

Summary

- You should now be able to:
  - write down the template for a list data definition
  - use structural decomposition to define simple functions on lists

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board

- Do Guided Practice 4.3

- Go on to the next lesson