Objectives

By the end of this lecture, you will know:

• What list recursion is

By the end of this lecture, you will be able to:

• follow the design recipe to write procedures that recur on lists

• draw a recursive diagram to help you write the recursive portion of a recursive procedure.

1 Announcements

• We’ve cut you some slack on homework handins during shopping period. No more. Dealing with this takes up valuable TA time that could be spent on TA hours with students.

  – Follow the instructions in the homework. If they’re unclear to you, use Piazza or email to the TAs to ask about them.

  – Text files are text files. PDFs, Word documents, RTF files, etc., are not text files. Text files are easy for us to anonymize, which is why we require them. We won’t grade anything else.
Handins for an assignment completely overwrite previous handins for that assignment. If you hand in problems 1, 2, 3, and then write a better answer for problem 2, and hand in problem 2, then your problem 1 and problem 3 handins will not be graded. *Always hand in the whole assignment.*

- The Artemis Project is a five week computer science camp for local 9th grade girls. Apply for a fun and rewarding summer job in computer science where you can make a difference while teaching what you love. See the link in the first slide for today’s class.

## 2 Lists: Two Program Interpretations

Recall, there are two kinds of lists: empty and cons-lists. This leads us to having two list predicates: $(\text{empty?} \ \text{empty}) \Rightarrow \text{true} \ (\text{cons?} \ (\text{cons} \ a \ b)) \Rightarrow \text{true}$ Also there are two main cons-list properties: $(\text{first} \ (\text{cons} \ a \ b)) \Rightarrow a \ (\text{rest} \ (\text{cons} \ a \ b)) \Rightarrow b$ Be sure to note that calling either first or rest on the empty list produces an error. For now, we will be using a printed representation of lists that shows how they were constructed. $(\text{cons} \ "\text{apple}\" \ \text{empty}) \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ \text{empty}))$ or just empty.

Now we are going to review the design recipe for things involving lists. One thing to note is that we no longer expect you to give examples of data types you are very comfortable with like nums, booleans, and strings. However, keep giving list examples until I can tell you are confident with them.

1. **Define the data**, specifically the type of list the procedure will process.

2. Provide **examples** of the data the procedure will process. Also, provide examples of the output data the procedure will produce.

3. Specify the procedure’s **type signature**, which describes the type of lists and other data the procedure consumes, and the type it produces.

4. Following the type signature, describe the procedure’s **call structure**. Doing so gives names to the procedure and its arguments.

5. Write a **specification** for the procedure. That is, in words, not code, state the relationship between the procedure’s input and output. Make sure to refer to the inputs by their argument names.

6. Provide **test cases** that exemplify the procedure’s operation. These tests must follow its call structure and satisfy its specification. *Be sure to test all cases in your data definition: i.e., all possible varieties of lists.*

7. Write a **template** for the procedure based on the data definition and type signature. [More on this below.]

8. **Code** the procedure, using the template as a reference. Specifically, fill in the template clause by clause. For each of the possible input types, decide which fields in the input structures are relevant to the problem at hand, and figure out how to operate on them to generate the desired output.

9. **Run** your program on your test cases.
What’s this “template” thing?

The template for a procedure that operates on lists follows the structure of the data by using a cond expression. Note, the structure of our program will mimic the structure of our data. You will almost always use cond when writing procedures with lists. And because lists come in two forms, our cond expression will always have two cases: one for the argument being an empty list, the other for when it’s a cons list. Those will be the two “conditions” in the cond-expression.

What about the two results? For the empty list, there’s a single result to be written down, and the empty list itself has no associated data except that it’s empty. So we write ...

For a cons-list, the result can depend on the contents of the list, which we can get by using the first and rest procedures to access that content. So the result, for the cons case of the cond expression, looks something like

\[
(\ldots \ (\text{first my-list}) \ \ldots \ (\text{rest my-list}) \ \ldots )
\]

Now let’s get to writing that program.

Step 1. The first step in the design recipe is to give an abstract definition of the data. As you know, our plan is to write a list-size procedure that operates on lists. But what sort of lists? For concreteness and homogeneity, let’s work with lists of numbers. In this case, we have:

`;; (num list)`
`;; - empty`
`;; - (cons num (num list))`

Step 2. Here are some examples of lists of numbers:

\[
(\text{define list0 empty})
(\text{define list1 (cons 19 empty)})
(\text{define list2 (cons 18 (cons 19 empty)})
(\text{define list3 (cons 17 (cons 18 (cons 19 empty)})
\]

Step 3. Write a type signature.

Step 4. Include the call structure

Note that alon is an abbreviation for “a list of numbers”.

Step 5. Next give the input and output specification.

Step 6. Following the data definition and examples provided in Step 1, construct test cases:

\[
\text{It is essential that you test all cases that arise in your data definition: i.e., all possible varieties of input data.}
\]

Step 7. The next step in the design recipe is to write a template for our procedure based on the data definition and type signature. Here’s how to do that for lists:
This abstract template describes how procedures typically operate on lists. Following the recursive data definition, the template considers two cases, one for each case of the data definition, in this case empty? and cons?. In the cons? case, where the input is non-atomic, the procedure uses the data’s selection operators to get at its constituents, namely the first element and the rest of the list.

At this point, we have completed six of the eight steps in our design recipe. We have a data definition and examples of the data, a type signature, a call structure, a specification, some test cases, and a template.

Step 8 The next (and certainly not least) step is to fill in our template.

To do so, we must determine two things: first, what should our procedure do in the empty? case; second, what should it do in the cons? case? To determine what it should do in the empty? case is simple: If the list is empty, the procedure should produce "small". So, we fill in "small" in the appropriate location in our template.

Now we are going to write the procedure list-size that consumes a list of nums and produces "small" for and empty list and "cons" for a cons list.

Now here is list-size in its entirety:
(check-expect (list-size lst2) "big")
(check-expect (list-size lst3) "big")

Note on check-expects is to remember to test the edge cases! For lists particular cases are the empty
list and the one element list. And always have one test case (at least) that isn’t edge case.

Step[9] But wait! There’s one more step in our design recipe: run. Does our code pass all of our test
cases? Normally, Racket will answer this question for you. And we showed in class that it worked
fine. But in general, we’ll have to evaluate things manually just to get a grip on what this recursion
thing is doing.

Something to think about is take the procedure application expression : (a b c) and what do you
think ”a” must be? One might think it should be a procedure but that is incorrect. Think of the
case ($(if (>(5 3)) + -)2 3) The if statements would be our ”a”, and it is not a procedure but
rather something that evaluates to a procedure.

3 Implementing a program

Now let’s try to implement another program using lists, list-length, which calculates a list’s
length.

Step[1] The first step, again, in the design recipe is to give an abstract definition of the data. As
you know, our plan is to write a list-length procedure that operates on lists. But what sort of
lists? For concreteness, let’s work with lists of integers. In this case, we have:

;; Data definition:
;; (num list)
;; — empty
;; — (cons number (num list))

Step[2]

(define list0 empty)
(define list1 (cons 19 empty))
(define list2 (cons 18 (cons 19 empty)))
(define list3 (cons 17 (cons 18 (cons 19 empty))))

Step[3] Write a type signature. As already mentioned, our list-length procedure will take as
input a list of integers. It will output a single integer:

;; list-length : (num list) → num

Step[4] The list-length procedure has the following call structure:

(define (list-length aloi)
  ... )

5
Step 5 The specification of the list-length procedure is straightforward:

```scheme
;; Input: a list of integers, aloi
;; Output: an integer, the length of aloi
```

Step 6 Following the data definition and examples provided in Step 1 construct test cases:

```scheme
;; Test cases for length
(check-expect (list-length list0) 0)
(check-expect (list-length list1) 1)
(check-expect (list-length list2) 2)
(check-expect (list-length list3) 3)
```

Again note that we’re testing on all cases that arise in the list data definition.

Step 7 The next step in the design recipe is to write a template for our procedure based on the data definition and type signature. Just like last time:

```scheme
(define (list-length aloi)
  (cond
    [(empty? aloi) ... ]
    [(cons? aloi) ( ... (first aloi) ... (list-length (rest aloi)) ... )]))
```

At this point, we have again completed six of the eight steps in our design recipe. We have a data definition and examples of the data, a type signature, a call structure, a specification, some test cases, and a template.

Step 8 The next (and certainly not least) step is to fill in our template. Just like last time, we must ask ourselves: first, what should our procedure do in the empty? case; second, what should it do in the cons? case? To determine what it should do in the empty? case is easy: check the test cases. The answer is immediate: If the list is empty (i.e., list0), the procedure should produce 0. So, we fill in 0 in the appropriate location in our template.

```scheme
(define (list-length aloi)
  (cond
    [(empty? aloi) 0]
    [(cons? aloi) ... (first aloi) ... (list-length (rest aloi)) ... ]))
```

## 4 Recursive Diagrams to the Rescue!

For the cons? case, we use a peculiarly clever idea. That is, the length of a compound list (cons item lst) is always 1 greater than the length of lst. To further illustrate this idea, we rely on a
neat tool—a *recursive diagram*— and help us break down the problem. You can write these before you start writing the code.

1. First, we write down the original input and the corresponding original output. The original input is whatever example you’re working on. The original output is whatever the procedure-specification says it should be. Filling in these two parts of the diagram doesn’t require anything but the specification.

2. Second, we figure out the recursive input (this is often already done in the template). For many many list-processing procedures, the recursive input is the “rest” of the list; it’s certainly a good thing to try. We’ll soon see other possibilities, however.

3. Third, we pose the problem on the recursive input, which yields the recursive output (again, by looking at the procedure specification).

4. Fourth, we figure out how to reconstruct the original output from the recursive output.

Here is a sample recursive diagram for the *list-length* procedure:

- original input: (cons 1 *empty*)
  - recursive input: *empty*  
  - recursive output: 0

- original output: 1

And here is another:

- original input: (cons 10 (cons 20 (cons 30 (cons 40 (cons 50 *empty*)))))
  - recursive input: (cons 20 (cons 30 (cons 40 (cons 50 *empty*))))
  - recursive output: 4

- original output: 5
If we try to do a recursion diagram with an empty list we can’t take the rest of one but that is okay because we know the output is already 0. In both cases, I’ve left a gap between these items; that’s space in which you can write ideas about how you might transform the recursive result into the original result. In the case of this procedure, that’s fairly easy, I hope. As you can see, we simply add one to get from our recursive output to our original output. In more complex procedures, this is where the real work happens.

So this approach only really requires that we figure out two key things:

1. how to derive the recursive input from the original input
2. how to derive the original output from the recursive output

Let us address each of these questions in turn. First, to derive the recursive input from the original input, we will take a closer look at the recursive data definition for lists. The recursive input is simply the smaller list of numbers, from which the original input is constructed. And how can we get at this smaller list? Using the `rest` selector, of course. In general, the most important feature of the recursive input is that it must be in some way smaller than the original input. We’ll talk about what “smaller” can mean in many ways over the next few weeks. But for many of the procedures that you’ll write for the next week, the way to get the recursive input is to compute the `rest` of the original input.

In the first diagram, we see that the length of the recursive input is 0, while the length of the original is 1. In the second diagram, we see that the length of the recursive input is 4, while the length of the original is 5. How can we derive the original output from the recursive output? Easy. The length of the original output is 1 more than the length of the recursive output. (So we might, in that open space between the two outputs, write something like “Just add one!”)

Hence, our code looks like this:

```racket
(define (list-length aloi)
  (cond
    [(empty? aloi) 0]
    [(cons? aloi) (+ 1 (list-length (rest aloi))))])
```

**Question:** Does it really make sense to call `list-length` on `rest aloi`? The `list-length` procedure operates on `aloi`.

**Answer:** In the call structure `(list-length aloi)`, `aloi` is just a name for `list-length`’s formal argument. When the `list-length` procedure is evaluated, the actual argument with which the procedure is called (in this case, `rest aloi`) is bound to the formal argument (in this case, `aloi`). This binding is then added to the environment under which the procedure is evaluated.

**Question:** What would happen if we were to recur on the whole of the input list instead of just the rest of that list?

**Answer:** Your input would never get smaller, so your procedure would never terminate. That would violate the advice that the recursive input should be smaller than the original input.

And we’ve done it! We’ve written our first *recursive* procedure in Racket! Here it is, in its entirety:
(define list0 empty)
(define list1 (cons 17 empty))
(define list2 (cons 18 (cons 17 empty)))
(define list3 (cons 19 (cons 18 (cons 17 empty))))

;; list-length : (num list) -> num
;; Input: a list of integers, aloi
;; Output: an integer, the length of aloi
(define (list-length aloi)
  (cond
   [(empty? aloi) 0]
   [(cons? aloi) (+ 1 (list-length (rest aloi)))]))

;; Test cases for length
(check-expect (list-length list0) 0)
(check-expect (list-length list1) 1)
(check-expect (list-length list2) 2)
(check-expect (list-length list3) 3)

Step 9: But wait! There’s one more step in our design recipe: run. Does our code pass all of our test cases? Yes! But why? For that, you can work through the rules of processing in the particular cases of list0 and list1 to see what happens, and stand back and admire the cleverness of the rules. If you don’t want to do that, I’ll be doing it for you on Monday.

For now, I want to skip the part about why it works, and practice writing more procedures like this one, which you’ll be doing again in lab and on this week’s homework.

Note: There is actually a length procedure built into Racket that does just what our list-length procedure does. You’re welcome to use it from now.

5 Additional list recursion problems

5.1 Improving a list

The improve procedure consumes an num list, and produces a new num list of the same size as the input, but with each item replaced by the number 17. (What a great improvement!)

Example:

> (improve (cons 1 (cons 3 empty))
There’s one design-recipe step that varies from our previous ones: the type-signature is improve: 
(num list) -> (num list), i.e., we’re returning a new type of thing (not a bool, as in size, nor 
an int, as in my-length. Here’s the solution:

```
;; Data definition
;; a num list is either
;; empty
;; (cons item b) where item is a num, and b is a num list
;; nothing else is a num list.
;;
;; examples:
;; num: 3, -4
;; num list:
(define 10 empty)
(define 11 (cons 1 empty))
(define 12 (cons 33 (cons -4 empty)))
;; improve: (num list) -> (num list)
;; input: aloi, a list of integers
;; output: a list of the same length, but with every input item replaced by 17
(define (improve aloi)
  (cond
   [(empty? aloi) empty]
   [(cons? aloi) (cons 17 (improve (rest aloi)))]))
(check-expect (improve empty) empty)
(check-expect (improve 11) (cons 17 empty))
(check-expect (improve 12) (cons 17 (cons 17 empty)))
```

In class, we derived this by writing out recursive diagrams

- original input: (cons 1 (cons 3 empty))
  - recursive input: (cons 3 empty)
  - recursive output: (cons 17 empty)

- original output: (cons 17 (cons 17 empty))

How can we derive the original output from the recursive output? We can cons 17 onto the recursive output. Therefore, in the recursive cons? case, we can write (cons 17 (improve (rest aloi)))
5.2 Looking for 17

Let’s do another:

`contains17?` takes as input an num list, and returns `true` if one of the items is the number 17, and `false` if the list does not contain a 17.

The type-signature is

```
contains17?: (num list) -> bool
```

To have good examples, we really need some example data with and without 17s in it, so that changes, too. Otherwise, a recursive diagram will help you figure out that the result, for a cons lists, should be true if the first item is 17 (obviously!) or if the rest of the list contains a 17. That begs for an or-expression.

And the code is

```
;; Data definition
;; a num list is either
;;  empty
;;  (cons item b) where item is a num, and b is an num list
;;  nothing else is a num list.
;;
;; examples:
;; num: 3, -4
;; num list:
(define 10 empty)
(define 11 (cons 17 empty))
(define 12 (cons 3 empty))
(define 13 (cons 33 (cons 17 empty))))
(define 14 (cons 33 (cons 17 (cons 5 empty))))
(define 15 (cons 33 (cons 18 (cons 5 empty))))

;; contains17?: (num list) -> bool
;; input: aloi, a list of integers
;; output: false if the list does not contain 17; true if it does.
(define (contains17? aloi)
  (cond
    [(empty? aloi) false]
    [(cons? aloi) (or (= (first aloi) 17)
      (contains17? (rest aloi)))]))
(check-expect (contains17? 10) false)
(check-expect (contains17? 11) true)
(check-expect (contains17? 12) false)
(check-expect (contains17? 13) true)
(check-expect (contains17? 14) true)
(check-expect (contains17? 15) false)
```
6 Summary

Ideas

- We’ve introduced the concept of lists in programming. Lists are an important data structure in Racket. In CS17, we will use homogeneous lists - lists of one type - in quite a few procedures. As such, we’ve also introduced a specialized design recipe when working with lists.

- We’ve also introduced the concept of recursion. Recursive procedures work by calling themselves. In functional programming, recursion offers an elegant way to solve problems.

Skills

- We’ve introduced the first and rest selector which access the first item in the list and a list containing all but the first item, respectively.

- We’ve also introduced two predicates. empty? checks if a list is empty or not, and cons? checks if a list contains any data or not.

- We’ve learned how to use recursive diagrams to break down and solve a problem. While the example of recursive diagrams for list size is primitive, you will soon see how to use recursive diagrams to solve complex problems.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)