(Provisional) Lecture 08: List recursion and recursive diagrams
10:00 AM, Sep 22, 2017

Contents

1 Announcements 1

2 Evaluation Correction 1

3 Lists 2
   3.1 Compound Lists ................................................................. 3
   3.2 Teasing apart cons-lists ....................................................... 4
   3.3 Homogeneous Lists .............................................................. 5
   3.4 List types .............................................................................. 5
   3.5 List Definition ....................................................................... 6

4 Lists: Two Program Interpretations 6

5 Testing and understanding our program 10

6 Recursive Diagrams to the Rescue! 12

7 Additional list recursion problems 15
   7.1 Improving a list ................................................................. 15

8 Summary 16

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 handin 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 Evaluation Correction

There is a small correction to the rules of evaluation for applying a user-defined procedure. The rules are:

1. Start from the top-level environment
2. Make the current environment be the TLE by binding the formal arguments in the closure to the actual arguments (by evaluating all other expressions in the procedure-application expression). The TLE remains unchanged
3. Evaluate the body in this current environment to get the value, which is the value of the entire procedure-application expression
4. Make the current environment be the TLE again.

The change here is that we must start from the top-level environment first.

Another important point to remember is that in a procedure application expression that looks like

\[((\textproc{<proc> } \textarg{<arg1> ... <argn> })\text{ )}\]

the first item \textproc{<proc>} doesn’t need to be a procedure, it just needs to evaluate to a procedure.

For example, in the expression

\[((\textif{ (> 5 3) + - ) 0 4)}\]

the first part \textif{( > 5 3) + -) is not just simply a name like + or - that evaluates to a procedure; it is an if expression, but it still evaluates to a procedure.
3 Lists

A Racket list is an abstraction of the kind of list you’re familiar with, like a shopping list or a class list, but with some restrictions. Intuitively, a list is a bunch of things in a row. We can abstract out essential components of a list to make it more applicable in more situations. Take Charlie Brown for example. As Scott McCloud said in his book “Understanding Comics,” Charlie Brown is drawn so generically so that more people can relate to him. The purpose of abstraction is to generalize.

In Racket, the items in a list can be of many different types (e.g. a boolean, then an int, then a boolean, then a string, ...). But in CS17, our lists will always be homogeneous: we’ll have a list of booleans, or a list of integers, or a list of strings, etc. Because of this, we can describe such a list, once we build one, with a new notation (for use in our type signatures):

```
(int list)
(bool list)
(str list)
```

There are exactly two kinds of lists: an empty list and a cons list. The empty list is just what it sounds like: it has no contents. A cons-list (“cons” is short for “construct”) actually has some contents. The empty list is used as a starting point for building up cons-lists.

There’s exactly one empty list, and it’s called empty. It is a keyword, which means we can’t use it as a name when creating our definitions, but it also acts as an expression. Note, an empty list is the abstraction for your grocery list before you add any groceries to it. As such, an empty list of numbers is your list before you add numbers to it and an empty list of strings is your list before you add the strings to it.

There’s an associated predicate, empty?, which is true for the empty list and false for everything else.

```
(empty? empty) => true
(empty? 15) => false
(empty? jar) => reference to an identifier before its definition: jar
```

The printed representation of the (sole) empty list is empty.

3.1 Compound Lists

The other kind of list is a nonempty, or compound list or cons list.

Here’s an example of building a “cons-list” with the builtin procedure cons:

```
(cons 31 empty)
```

As you can see from the example, the cons procedure is used in this format: (cons item list).
In this example, the number 31 is in the first field and the list empty is in the second. That is to say, a cons-list structure has two parts: a datum and a list. The datum represents a new item to be added to the list that is the second item, and the value is (intuitively) a new list that contains that item followed by all the items of the second list in their original order.

To be useful, such a cons-list must not only be able to be constructed, but should be able to be deconstructed as well: we should be able to somehow discover what items are in the list. We’ll come to that in just a moment.

Terminology: Compound lists or cons lists are constructed by “cons”-ing the first field onto the second.\(^1\)

Here’s another example:

\[
\text{(cons 18 \ (cons 31 \ empty))}
\]

One essential feature of a compound list is: the second field is required to be a list: either an atomic list (i.e., empty), or another compound list. So, this is a compound list: \((\text{cons 17 \ (cons 18 \ empty))}\). But this is not: \((\text{cons 17 18)}\). In fact, in CS17, that last expression is not even allowed. (In some versions of Racket, it is allowed; read about those on your own if you’re interested, but never use them in CS17.)

You can use the predicate \(\text{cons?}\) to test whether a datum is a compound list.

\[
\begin{align*}
(\text{cons? \ (cons 17 \ empty))} & \Rightarrow \text{true} \\
(\text{cons? \ 15)} & \Rightarrow \text{false} \\
(\text{cons? \ empty}) & \Rightarrow \text{false} \\
(\text{cons? \ empty?}) & \Rightarrow \text{false}
\end{align*}
\]

We can also see the predicate \(\text{empty?}\) return \text{false} now.

\[
(\text{empty? \ (cons 3 \ empty))} \Rightarrow \text{false}
\]

### 3.2 Teasing apart cons-lists

The two things that constitute a compound list can be accessed using the \text{first} and \text{rest} selectors.\(^2\)

---

1. “cons” is short for “construct.”
2. In the “olden days,” \text{first} and \text{rest} were named \text{car} and \text{cdr}, respectively. CAR stood for Contents of Address Register, and CDR, for Contents of Decrement Register. Although you may encounter these terms from time to time in your CS career (indeed DrRacket recognizes these names), the corresponding descriptions are no longer accurate, and we’ll never use them in CS17.
(first (cons 17 empty))
=> 17

(rest (cons 17 empty))
=> empty

More generally, the first and rest selectors satisfy the following algebraic identities:

(first (cons x y))
=> x

(rest (cons x y))
=> y

These selectors produce an error if their argument is not a compound list:

(first 17)
=> first: expected argument of type <non-empty list>; given 17

(rest empty)
=> rest: expected argument of type <non-empty list>; given empty

**Question:** Can the first field store a list as well?

**Answer:** Sure! The first field can store any kind of data. If a list’s first field is a list of numbers, for example, then the list itself is a list of list of numbers.

**Question:** Are there any selectors that can access any element of a list (not just the first element of a list and the rest of the list)?

**Answer:** Unfortunately not. You will have to rely on first and rest to access those elements. You will learn how to do that (very) soon!

### 3.3 Homogeneous Lists

Though we sometimes fail to be rigorous about this point—in particular, in the lecture thus far, we spoke of lists generically—there are, for CS 17, only lists of numbers, lists of booleans, lists of strings, lists of structs, etc. There are also lists of lists of numbers, lists of lists of lists of numbers, and so on.

Here are formal definitions of a few representative lists:

A list of strings is either:

- empty, or
- (cons str alos), where str is a string and alos is a list of strings

Nothing else is a list of strings.

Similarly, a list of numbers is either:
• **empty**, or
  • (cons num alon), where num is a number and alon is a list of numbers

Nothing else is a list of numbers.

As we have already noted, lists are containers—containers for data. In CS 17, we insist that the lists you define be **homogeneous**. This means that they can contain only one type of data. So lists of strings are fine, and lists of numbers are fine, but please do not create lists that contain both strings and numbers (or any other combination of non-homogeneous data).

In particular, the following sort of list is a no-no in CS 17:

```lisp
(define i-am-not-a-CS-17-list (cons 15 (cons "fifteen" empty)))
```

### 3.4 List types

If we have a procedure that operates on a list of integers and produces a string, we write

```lisp
;; my-proc: (int list) -> str
```

That is to say: lists are described (in signatures) in the form (... list), where the ellipsis is some data type and the parens and the word “list” are required.

We’ll return to this and make it a bit fancier soon.

### 3.5 List Definition

The above definitions, in which a datatype (here, a list) is defined in terms of itself, is called **recursive**, or **inductive**. Note that well-crafted recursive definitions are not circular because: (i) there is a **base case**, in which the type is not defined in terms of itself, and (ii) the other parts in the recursive definition define the data of that type in terms of a “smaller” datum of that type.

### 4 Lists: Two Program Interpretations

Recall that at the end of last class, we wrote a procedure called `list-size` that returns the string "small" when a-list is empty and "big" otherwise. To make it concrete, we’ll have it operate on `int` lists.

To write this procedure, we’ll rely on our trusty design recipe, slightly revised:

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 cons expression, looks something like

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

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:

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

Step[2] Here are some examples of lists of numbers:

```scheme
(define list0 empty)
(define list1 (cons 19 empty))
(define list2 (cons 18 (cons 19 empty)))
(define list3 (cons 17 (cons 18 (cons 19 empty))))
```
Here are some examples of strings:

"abc"
"big"
"small"

Step 3 Write a type signature.

As already mentioned, our list-size procedure will take as input a list of integers. It will output a string that is either "small" or "big".

;; list-size : (int list) → string

Step 4 The length procedure has the following call structure:

(define (list-size aloi)
  ...
)

Note that aloi is an abbreviation for “a list of integers”.

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

;; Input: a list of integers, aloi
;; Output: a string, "small" if the input list is empty, "big" otherwise.

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

;; Test cases for list-size
(check-expect (list-size list0) "small")
(check-expect (list-size list1) "big")
(check-expect (list-size list2) "big")
(check-expect (list-size list3) "big")

It is essential that you test all cases that arise in your data definition: i.e., all possible varieties of input data. Here we’ve checked every variety of input data, both empty and cons lists!

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:

(define (list-size aloi)
  (cond
   [(empty? aloi) ... ]
   [(cons? aloi) ( ... (first aloi) ... (list-size (rest aloi)) ... )]]))

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.

The cons? case for this example is equally simple. If the list is a compound list, return "big". Hence, our code looks like this:

```
(define (list-size aloi)
  (cond
    [(empty? aloi) "small"]
    [(cons? aloi) "big"]))
```

**Question:** What happened to (first aloi) and (rest aloi)? It was part of the template, but it isn’t part of the final procedure.

**Answer:** As it happens, the list-size procedure doesn’t make use of the first element of the input list nor the rest of the list. In general, these the only things you can use, but you don’t have to use all of them! Other procedures however, as you will see later, do use first and rest. All constituents of the data must appear as part of the template, even though not all procedures need access to all constituents, because you may not know beforehand exactly which constituents are needed by which procedures.

Now here is list-size in its entirety:

```
;; (int list)
;; - empty
;; - (cons number (int list))

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

;; example of strings
;; ""
;; "small"
;; "big"

;; list-size : (int list) → string
;; Input: a list of integers, aloi
```
;; Output: a string, "small" if the input list is empty, "big" otherwise.

(define (list-size aloi)
  (cond
   [(empty? aloi) "small"]
   [(cons? aloi) "big"]))

;; Test cases for list-size
(check-expect (list-size lst0) "small")
(check-expect (list-size lst1) "big")
(check-expect (list-size lst2) "big")
(check-expect (list-size lst3) "big")

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.

5 Testing and understanding our program

So far everything’s been review from last class. Here’s something new.

Let’s follow through what happens when we evaluate

(list-size lst0)

We first identify that this is a procedure application expression. We begin by looking up list-size in the top-level environment, and see that it is bound to a closure, with the formal argument aloi and a cond expression as its body. We must then evaluate lst0, an identifier; looking up in the environment, we see that it is bound to the empty list. Because this is a procedure application, the body of the closure is now evaluated under a new, augmented environment containing aloi bound to the empty list. For visual simplicity, we can substitute list-size with the cond expression body, and all instances of aloi with the empty list.

(list-size lst0)
=> (cond
   [(empty? empty) "small"]
   [(cons? empty) "big"])

Finally, we can evaluate this cond expression by going through each case in order, as we did in Lecture 04. Because the condition in first case evaluates to true, this cond expression will evaluate to the string "small".

(cond
  [(empty? empty) "small"]
  [(cons? empty) "big"]
=> "small"

Hence, our first test passes.

Second,
We begin by evaluating `list-size` the same way we did above; it evaluates to the same closure. We next evaluate `lst1`, another identifier. We look it up in the environment, and see that it is bound to `(cons 19 empty)`. The body of the closure is now evaluated under a new, augmented environment containing `alon` bound to `(cons 19 empty)`. For visual simplicity, we can once again substitute `list-size` with the `cond` expression body, and all instances of `alon` with the list `(cons 19 empty)`.

```
(list-size lst1)
```

```
=> (cond
    [(empty? (cons 19 empty)) "small"]
    [(cons? (cons 19 empty)) "big"])
```

We once again evaluate the `cond` expression by going through each case in order. Clearly, `(cons 19 empty)` is not an empty list, so the condition in the first case evaluates to `false` and we move on. Looking at the second case, the condition evaluates to `true` because `(cons 19 empty)` is a non-empty list. The `cond` expression therefore evaluates to the string "big".

```
(cond
    [(empty? (cons 19 empty)) "small"]
    [(cons? (cons 19 empty)) "big"])
=> "big"
```

Hence, our second test also passes. You can verify the other test cases on your own!

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:

```
;; (int list)
;; - empty
;; - (cons number (int 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))))
```

Here are some examples of integers:

```
;; output examples
;; 0
;; 1
```

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:
### Step 4
The `list-length` procedure has the following call structure:

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

### 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) 0]
   [(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)) ... )])
```
6 Recursive Diagrams to the Rescue!

For the \texttt{cons?} case, we use a peculiarly clever idea. That is, the length of a compound list (\texttt{cons item lst}) is always 1 greater then the length of \texttt{lst}. To further illustrate this idea, we rely on a neat tool—a \textit{recursive diagram}— and help us break down the problem:

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 \texttt{list-length} procedure:

- original input: \texttt{(cons 1 (cons 2 (cons 3 \texttt{empty})))}
  - recursive input: \texttt{(cons 2 (cons 3 \texttt{empty})�)}
  - recursive output: 2

- original output: 3

And here is another:

- original input: \texttt{(cons 10 (cons 20 (cons 30 (cons 40 (cons 50 \texttt{empty})))))}
  - recursive input: \texttt{(cons 20 (cons 30 (cons 40 (cons 50 \texttt{empty}))))�}
  - recursive output: 4
• original output: 5

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 2, while the length of the original is 3. 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:
;;; (int list)
;;; - empty
;;; - (cons int (int list))

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

;;; example ints
;;; 0
;;; 1

;;; list-length : (int list) -> int

;;; 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.

7 Additional list recursion problems

7.1 Improving a list

The improve procedure consumes an int list, and produces a new int 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)))
(cons 17 (cons 17 empty))
There’s one design-recipe step that varies from our previous ones: the type-signature is `improve: (int list) -> (int 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
;;; an int list is either
;;; empty
;;; (cons item b) where item is an int, and b is an int list
;;; nothing else is an int list.
;;; examples:
;;; int: 3, -4
;;; int list:
;;; (define l0 empty)
;;; (define l1 (cons 1 empty))
;;; (define l2 (cons 33 (cons -4 empty)))
;;;
;;; improve: (int list) -> (int 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 l1) (cons 17 empty))
(check-expect (improve l2) (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 case, we can write `(cons 17 (improve (rest aloi)))`
8 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 CS17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs017/feedback