Lecture 07: Lists

10:00 AM, Sep 20, 2017

Contents

1 Another Look at Conditionals .................................................. 1
2 Lists ......................................................................................... 2
   2.1 Compound Lists ................................................................... 3
   2.2 Teasing apart cons-lists ......................................................... 4
   2.3 Homogeneous Lists ............................................................... 4
   2.4 List types ............................................................................. 5
   2.5 List Definition ...................................................................... 5
3 Lists: A Program Interpretation .................................................. 5
4 Summary ..................................................................................... 8

Objectives

By the end of this lecture, you will know:

• what lists are
• how to construct them

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

• use conditionals to minimize repeated code
• follow the design recipe to set up procedures that involve lists

1 Another Look at Conditionals

When we write a program, we want to minimize our mistakes. What is one way to do this? By using as little repeated code as possible. Think about it – the fewer times you need to write something, the less likely you are to make a mistake or create a discrepancy.

For example, let’s say we are walking, and we come to a fork in the road. Which path should we take? Well, given some very informative signs, we know that going left, there’s “quicksand, cliffs, and snakes ahead”. Going right, there’s “quicksand, cliffs, lions, tigers, bears, and snakes ahead”. That’s a lot of repeated text! In Racket, how could we streamline it?
We’ve already seen some uses of conditional statements, namely if and cond expressions. Well, there is a way we can use conditionals effectively in our scenario, in conjunction with the built-in procedure string-append.

The string-append procedure consumes multiple strings, and combines them into one. For example:

```
(string-append "see" "saw") ; outputs "seesaw"
(string-append "I " "love " "dogs") ; outputs "I love dogs"
```

Note the use of spaces in the latter example.

Now, let’s see how we can approach our initial scenario, assuming left and right are our only two options:

```
(define (which-way direction)
  (string-append "Warning! Quicksand, cliffs, "
    (if (string=? direction "left") "" "lions, tigers, bears, ")
    "and snakes ahead!"))

(which-way "left") ; outputs "Warning! Quicksand, cliffs, and snakes ahead!"
(which-way "right") ; outputs "Warning! Quicksand, cliffs, lions, tigers,
                       bears, and snakes ahead!"
```

Using an if-expression, we were able to minimize our repeated code (and we’d probably decide that going left is the lesser of two evils).

## 2 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.

\[
\begin{align*}
\text{(empty? } \text{empty}) &= \text{true} \\
\text{(empty? 15)} &= \text{false} \\
\text{(empty? jar)} &= \text{reference to an identifier before its definition: jar}
\end{align*}
\]

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

### 2.1 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`:

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

As you can see from the example, the `cons` procedure is used in this format: `(cons item lst)`. 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: `(cons 17 (cons 18 empty))`. But this is not: `(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 `cons?` to test whether a datum is a compound list.

\(^1\)cons is short for “construct.”
We can also now see the predicate empty? sometimes returns false:

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

2.2 Teasing apart cons-lists

The two things that constitute a compound list can be accessed using the first and rest selectors:

\[
\text{(first (cons 17 empty))} \\
\Rightarrow 17
\]

\[
\text{(rest (cons 17 empty))} \\
\Rightarrow \text{empty}
\]

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

\[
\text{(first (cons x y))} \\
\Rightarrow x
\]

\[
\text{(rest (cons x y))} \\
\Rightarrow y
\]

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

\[
\text{(first 17)} \\
\Rightarrow \text{first: expected argument of type <non-empty list>; given 17}
\]

\[
\text{(rest empty)} \\
\Rightarrow \text{rest: expected argument of type <non-empty list>; given empty}
\]

---

2 In the “olden days,” first and rest were named car and 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.
**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!

### 2.3 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)))
```

### 2.4 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.

2.5 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.

3 Lists: A Program Interpretation

Let’s start using lists in a program. We’ll start with a fairly silly example, 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 cond expression, looks something like

```
( ... (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:

```
(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:

```scheme
;; 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:

```scheme
;; 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:

```scheme
(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. 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, which we will go over next lecture when we talk about recursion (see next set of notes for information about recursion).

Step 9 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. But in general, we’ll have to evaluate things manually. We’ll do that in the next class.

4 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.
Skills

- We’ve introduced the `first` and `rest` selectors 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.

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](http://cs.brown.edu/courses/cs017/feedback)