Lecture 07: Lists
10:00 AM, Sep 18, 2019

Contents

1 New rules of processing 1
  1.1 and / or expressions ............................... 1
  1.2 if expressions .................................... 2

2 Another Look at Conditionals 2

3 Not 3

4 Lists 3
  4.1 Compound Lists .................................... 4
  4.2 Teasing apart cons-lists ............................ 5
  4.3 Homogeneous Lists .................................. 6
  4.4 List types .......................................... 7
  4.5 List Definition ..................................... 7

5 Summary 7

Objectives

By the end of this lecture, you will know:

• what lists are
• how to construct them
• how to use the built-in keyword not

You will also get some additional practice with and and or.

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 New rules of processing

1.1 and / or expressions

1. An and-expression \((\text{and}\ exp1\ exp2\ \ldots\ expn)\) is evaluated by evaluating \(exp1\), whose value must be a boolean. If it is \text{false}, the value of the and-expression is \text{false} and the remaining expressions are never evaluated. If it is \text{true}, however, we evaluate \(exp2\); again, the value must be a boolean or it’s an error. If the resulting value is \text{false}, the value of the and-expression is \text{false} and the remaining expressions are never evaluated. We proceed similarly through any remaining expressions. If all expressions evaluate to \text{true}, then the value of the and-expression is \text{true}.

2. An or-expression \((\text{or}\ exp1\ exp2\ \ldots\ expn)\) is evaluated by evaluating \(exp1\), whose value must be a boolean. If it is \text{true}, the value of the or-expression is \text{true} and the remaining expressions are never evaluated. If it is \text{false}, however, we evaluate \(exp2\); again, the value must be a boolean or it’s an error. If the resulting value is \text{true}, the value of the or-expression is \text{true} and the remaining expressions are never evaluated. We proceed similarly through any remaining expressions. If all expressions evaluate to \text{false}, then the value of the or-expression is \text{false}.

1.2 if expressions

An if-expression \((\text{if}\\ test\-\exp\ \text{true}\-\exp\ \text{false}\-\exp)\) is evaluated by first evaluating \(test\-\exp\) to produce a value \(b\); if the value’s not a boolean, it’s an error. If the value \(b\) is \text{true}, then the value of the if-expression is the result of evaluating \text{true}\-\exp, and \text{false}\-\exp is never evaluated; if the value of \(b\) is \text{false} then the value of the if-expression is the result of evaluating \text{false}\-\exp and \text{true}\-\exp is never evaluated. Also note that while Racket will allow the values of the \text{true}\-\exp and \text{false}\-\exp to be different (that is to say \text{true}\-\exp could evaluate to a number while \text{false}\-\exp could evaluate to a string, for example), this is not how we will write code in CS17. The values returned by the evaluation of the \text{true}\-\exp and \text{false}\-\exp will be of the same type for this course.

The idea that in evaluating each of these three, \text{and}, \text{or}, \text{if} (as well as \text{cond}), the rules involve \textit{not} necessarily evaluating all the expressions that appear after the keyword as mentioned before is called “short-circuiting”, and is amazingly helpful in some situations.

Of these four, \text{cond} is certainly the one you’ll use most in CS17. It’s so important that in our next language, Ocaml, they developed special syntax so that you don’t even have to write a word like “cond” in many cases.

It is important to keep in mind that \textit{if, and, or and cond} are keywords, not procedures. Additionally, they can not be redefined or used as arguments in procedures.

2 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).

### 3 Not

At this point in class, Spike asked students to write a procedure named not, which takes in an expression that evaluates to a boolean and returns the other boolean. For instance, something like (not (= 7 3)) would return true. Here is the code:

```
(define (not b)
  (if b false true))
```

Wait! If you put this into DrRacket and run it, it gives you an error! What’s wrong? Well, it turns out that not is actually a pretty handy built-in in Racket that you do not need to write yourself.

### 4 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):

\[
\begin{align*}
(int \ list) \\
(bool \ list) \\
(str \ list)
\end{align*}
\]

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.

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

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

4.1 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.

Here’s another example:

```
(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.

```
(cons? (cons 17 empty)) => true
(cons? 15) => false
(cons? empty) => false
(cons? empty?) => false
```

We can also now see the predicate `empty?` sometimes returns `false`:

```
(empty? (cons 3 empty)) => false
```

Q: use cons to construct the following list starting with the empty list: (1 2 3)

4.2 Teasing apart cons-lists

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

---

1 cons is short for “construct.”

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
(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!

### 4.3 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:

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.
• `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:

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

Q: which of these lists are homogeneous: (1 2 3) ("one" "two" "three") ("1" "two" "three")

4.4 List types

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

```
;; 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.

4.5 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.

5 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 \texttt{first} and \texttt{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. \texttt{empty?} checks if a list is empty or not, and \texttt{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 CS 17 document by filling out the anonymous feedback form: \url{http://cs.brown.edu/courses/csci0170/feedback}. 