Introduction

Bookmark this page

In the first four modules of this course, we've already managed to build up a substantial arsenal of programming tools and techniques, and we've introduced the Design Recipe in order to manage the complexity of using them.

Despite all this effort, however, we're constrained by a fundamental *finitude*: the programs we execute are bounded in the amount of work they are able to do. Think about it: you write a function, which may call other functions, which may call other functions, which ultimately bottom out in the use of built-in functions and special forms. This hierarchy imposes an upper bound on the amount of work you can do in terms of the amount of code you write.

And yet, in many real-world situations, we don't know in advance how much work we're going to need to do. Let's say, for example, that we're writing a program to manage information about students taking a course. One function we'd like to write consumes the students' exam grades and calculates their average. If we know we have exactly, say, five students, we can easily write such a function:

```
1 (define (exam-average m1 m2 m3 m4 m5)
2   (/ (+ m1 m2 m3 m4 m5) 5))
```

What happens if a sixth student joins the course?

```
1 (define (exam-average m1 m2 m3 m4 m5 m6)
2   (/ (+ m1 m2 m3 m4 m5 m6) 6))
```

And then what if there are twenty students in the course? Or hundreds? Do I really need to rewrite my program for each distinct number of students? And where would I even *put* these exam marks in the first place? Do I have to define hundreds of constants to hold them? Programming can't be so painful, can it? (Spoiler: no!)

As another quick example, consider the factorial function $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$. How would we write a Racket function `(factorial n)`? It seems here like the amount of work we'll need to do is tied directly to `n` itself. But we don't yet have the conceptual tools to make the amount of work we do stretch out to accommodate problems of different sizes.

In this module, we break decisively through this barrier, and see how to write small programs that can work their way through any amount of data. The key breakthrough is to introduce a new type of data, the *list*. A list is a kind of **compound value**, an object that Racket treats as a single value but that nevertheless contains other values inside itself. Imagine a shopping list—you write "milk, eggs, cheese" on a piece of paper, then fold up the paper and put it in your pocket. While it's folded up, you can think of the list as a single entity. It's only later, at the store, where you need to go "into" the list and retrieve the individual pieces of information it contains.

Lists will motivate us to roll out a lot of new ideas that interact in elegant and powerful ways. First, we'll learn the basic tools: the built-in functions that Racket provides for constructing and deconstructing lists. Then, when we start to write functions that operate on lists, we'll discuss **data-directed design** as a way of thinking about organizing functions based on the data they process. In the case of lists, that will lead us in a natural way to **recursion** as a programming technique. Recursion is often a stumbling block for students first learning to program. The good news is that recursion is introduced in this course in a very specific way that helps us keep it under control.

Lists and recursion have far-reaching consequences, which cannot be sorted out in this module alone. We'll spend the next three modules examining lists, recursion, and data-directed design from lots of different angles, before moving on to other topics.

Topics:

- Basic list functions
- Data-directed design
- Data definitions and template functions for lists
- List processing and recursion
- List idioms: mapping, filtering, and folding
- Variations on list processing: auxiliary data, non-empty lists

Discussion

Hide Discussion

Topic: Module 05 / Introduction

Add a Post

Show all posts | by recent activity

There are no posts in this topic yet.