05.2 Data-Directed Design

Bookmark this page

We now know that we can create lists of any length, containing any combination of values. How can we write a function that consumes a list of any length? For example, how would we write a function `sum-list` that consumes a list of numbers `lon` and produces the sum of all the numbers in the list?

```
> (sum-list (cons 3 empty))
3
> (sum-list (cons 3 (cons 6 empty)))
9
> (sum-list (cons 3 (cons 12 (cons 83765 (cons 1283795 empty)))))
1367575
```

Really, we don't yet have the necessary practical or intuitive tools we need to solve this problem (though you should feel free to try it as an exercise!). In order to build up to functions like `sum-list`, we're first going to pause and introduce some new machinery to the Design Recipe. As always, the additions to the Design Recipe will give us new organizational tools that help us stay on track when solving coding problems, without getting lost or confused. In this lesson we'll learn about the principle of **data-directed design**, and add **data definitions** and **function templates** to the Design Recipe as an expression of data-directed design.

Inventing new types

Let's revisit a function we encountered in Module 03:

```
(define (reciprocal x)
  (cond
    [(equal? x 'infinity) 0]
    [(equal? x 0) 'infinity]
    [else (/ 1 x)]))
```

The idea behind this function was to work with a kind of extended number system, consisting of regular numbers augmented by the symbol `'infinity`. That's an intuitive enough idea, but if we attempt to wrap the Design Recipe around this function, we immediately hit a wall: what's the contract for this function? In particular, what's the type of the argument `x`, and of the value produced by the function? It isn't `Num` or `Sym`, because the function can work with both. As a workaround, we could throw up our hands and revert to the catch-all type `Any`:

```
;; reciprocal: Any -> Any
(define (reciprocal x)
  ...)
```

This is very unsatisfying, because it leaves us with the form of a contract but none of the power to be selective about types. It's also inaccurate: whereas a function like `number?` really can operate on any value at all, our `reciprocal` function is bound more tightly to `Num` and `Sym`. If you try to pass in, say, a `Str`, you'l get an error. There's got to be a better way, a way to talk about types in between `Num` and `Any`. So let's invent one.

A type is just a set of values. When we want to use a set that doesn't correspond to an existing Racket type, we'll just write down a definition for that set, give the set a name, and use that name in the Design Recipe as if it were any other type. We just need some new standardized language to describe these new types. In a first pass, we put the following at the top of our program:

```
;; An ExtNum is one of:
;; * A Num
;; * A Sym
```

We refer to this comment as a **data definition**. Like a constant definition, it adds a new name to our universe, in this case `ExtNum`. We'll use a bulleted list format when a new type consists of values taken from a small set of options, here `Num` or `Sym`. Viewed in terms of set theory, the "one of" format suggests that the new type `ExtNum` is the *union* of `Num` and `Sym`.

We can further refine this data definition. The problem is that `reciprocal` understands only numbers and `'infinity`, and not any other symbols, so our definition of `ExtNum` is still loose. In addition to taking unions of existing types, let's also allow unions involving *individual values*. We'll write this:

```
;; An ExtNum is one of:
;; * A Num
;; * 'infinity
```

Ah, that's better. `ExtNum` is now precisely the type we want: all number values, together with the singleton `'infinity`. We now have the perfect lean-and-mean type to use in the contract of our function:

```
;; reciprocal: ExtNum -> ExtNum
(define (reciprocal x)
  ...)
```

In general, we'll write a data definition any time we'd like to talk precisely about a set of values that doesn't already have a convenient name in Racket. If we write a data definition, we usually include it at the top of a program, before any constants or functions—it applies to the whole program, and not to any individual function definition.

In some cases, we'll want to have a lightweight ability to talk about these sorts of union types, without the overhead of a full data definition. In that case, we invent the notation `(anyof` t_1 ... t_n v_1 ... v_m`)`, where each t_i is the name of a known type and v_i is a literal Racket value. This `anyof` shorthand gives us an "inline" way to talk about these types, skipping the data definition:

```
;; reciprocal: (anyof Num 'infinity) -> (anyof Num 'infinity)
(define (reciprocal x)
  ...)
```

The long-form data definition and the `anyof` notation are both valid. As with other aspects of programming, deciding which one to use is a matter of style and judgment.

It's worth being very clear on one point here: *Racket has no idea what we're doing!* Racket doesn't know about the type `ExtNum`, and `anyof` isn't a built-in function or special form. Data definitions are comments (or appear inline in contracts, which are also comments). Although we'll use a rigorous syntax for data definitions, they're purely for documentation purposes, a way to clarify the intent of and limitations on a value.

As usual, our language of data definitions is provisional. Several times during the rest of the course, we'll add new kinds of data definitions as we explore richer Racket types.

Concept Check 05.2.1

Racket doesn't know about the type `ExtNum`, and so it doesn't give us anything like a type predicate for this type. But we can write one ourselves.

Write a predicate `extnum?` that consumes a Racket value of any type and produces `true` if and only if the value belongs to the type `ExtNum`, defined as `(anyof Num 'infinity)`.

1/1 points
Attempts: 2 / Unlimited

```
(define (extnum? val)
  (cond
    [(number? val) true]
    [(and (symbol? val) (symbol=? val 'infinity)) true]
    [else false]))
```

Submit Code Reset Code

✔ Correct! 1/1 points

Concept Check 05.2.2

1/1 point (graded)

In Zen Buddhism, the word "mu" is sometimes used as a way of rejecting the premise of a question (or perhaps, of "unasking" the question). Let's define a new type `ZenBool`, short for "Zen Boolean", which consists of the two ordinary Boolean values together with the string `"mu"`.

Which of the following is a valid description of the type `ZenBool`?

- ○ `Any`
- ○ `(anyof Bool Str)`
- ○ `(anyof Bool Sym)`
- ● `(anyof true false "mu")`
- ○ `(anyof "true" "false" "mu")`

✔

Save Show answer

Submit You have used 1 of 2 attempts

Data-directed design

So far, it looks like we rolled out data definitions purely for bookkeeping purposes: they allow us to name new types, and use those names fluidly in the Design Recipe.

But data definitions are vastly more powerful than that. As we'll see, the work invested in writing down a data definition can streamline the entire rest of the coding process!

Let's have another look at the data definition we've been working with:

```
;; An ExtNum is one of:
;; * A Num
;; * 'infinity
```

What can we say about a generic function that consumes an `ExtNum`? We can't say much, because we don't know exactly what the function is supposed to do, but we can say *something*. Every `ExtNum` is one of two things: a number, or the symbol `'infinity`. In order to handle all `ExtNum`s, our hypothetical function will almost certainly have to figure out what kind of value it's got, so that it can decide whether to do "numbery" things or "symboly" things to it. So even if we squint and blur out the details specific to the problem we're solving, we'll probably need a `cond` that branches to one of two possible expressions:

```
(define (            val)
  (cond
    [(number? val)                ]
    [(equal? val 'infinity)                ]))
```

(As usual, we could simplify the `cond` by replacing the second question with an `else`, but let's not worry about that here.)

Notice the parallels between the data definition and the code. We can interpret the "is one of" as a hint that we're going to need a `cond`, in order to decide which "one of" we've got. Then, each case in the data definition is associated with a question/answer pair in the `cond`, using a question that exactly unravels the type described in the data definition's bullet:

```
                               (define (            val)
;; An ExtNum is one of:          (cond
;; * A Num                         [(number? val)
;; * 'infinity                     [(equal? val 'infinity)
```

This example demonstrates a far more general principle that we call **data-directed design**. This principle can be summarized as follows:

Data-directed design
The structure of the code should mirror the structure of the data.

That is, when we're trying to solve a problem that involves processing a value of a particular type, we should examine the way that type is assembled. That will guide us in the high-level structure of the function we're writing. We'll still have to fill in the details tied to the specific problem we're solving, but data-directed design will give us a running start and help us avoid getting lost.

Template functions

DrRacket doesn't let us write code with blurry text like in the diagrams above, but we can still try to capture the generic structure of a function to process a particular type of data, like the two questions used in the `cond` when we consumed an `ExtNum`. To that end, when we write down a data definition, we follow it up with a **template function** for processing values of the newly defined type. The template function isn't executable Racket code, but it lays out the general form of code we expect to write. Here's how we might write a template for processing an `ExtNum`:

```
;; extnum-template: ExtNum -> Any
(define (extnum-template n)
  (cond
    [(number? n) ...]
    [else ...]))
```

Notice the following rules, which we'll always follow:

1. We name the function *`typename`*`-template`, where *`typename`* is the name of the type we've defined, written in lowercase.
2. We give a contract that consumes a *`typename`* and produces `Any`, since we don't have any more specific information right now. We'll modify the contract later as needed.
3. Whenever there's a part of the template body where we don't know what we're going to do, we simply write in `...` (here, the dots *really are* in the template function, they're not a shorthand for something else that we're leaving out for brevity).

If a data definition's job is to tell you how to put together new values from old ones, then it's the template's job to tell you how to take values apart again. The template should *drill down*: go as deep as possible into the consumed value and lay all its contents out in front of you, so that you choose what to do with them later. In the case of an `ExtNum`, that's a fairly simple process: the type is assembled from two other types, so we decide what to do based on which of the two cases we've got. Later, the idea of drilling down will guide us in crafting more interesting template functions.

When we get to the point of writing actual functions, we will replace the `...` placeholders with code. We may also remove whole `cond` clauses if they're not needed, or make other changes (like adding an extra parameter to the function). The template is just a starting point, but a useful one.

> In practice, we don't require you to include template functions in the code you submit for assignments, though you're welcome to do so. But it's valuable always to write them first, before refining them into the solutions to programming problems. If you're trying to solve a problem using code that doesn't follow a template (or, indirectly, that doesn't adhere to data-directed design), there's a good chance you're on the wrong track and should seek a more elegant approach.

Eventually, we'll reach a point where templates start to pay diminishing returns: some of our data processing scenarios will be complex enough that the templates become too cumbersome to write and maintain. Hopefully by then you'll be more comfortable writing functions, enough to get by without a template for everything. Even later, we'll see that in many cases there are more advanced function programming features that allow us to skip the need for templates altogether.

Concept Check 05.2.3

1/1 point (graded)

Why does the contract of a template function always produce the type `Any`?

- ○ Because templates are only useful when writing functions that produce `Any`.
- ● Because when we write the template, we don't have enough information about the value produced by the function we'll write later based on this template.
- ○ Because `Any` comes first in an alphabetical listing of types in Racket.
- ○ Because you're allowed to write any word at all to the right of the arrow in a contact, and it doesn't affect the contract's meaning.

✔

Save Show answer

Submit You have used 1 of 2 attempts

Summary

Data-directed design is a way of thinking about programming, in which the form of the code we write parallels the data we're writing it to handle. We express these parallels by giving a data definition that explains how to assemble values of a new type, together with a template function that demonstrates how to take those values apart again.

We consider these constructions as additions to the Design Recipe, in a new preliminary step called **data analysis**: we think about the problem domain, identify any new types that would help in writing functions in that domain, and give data definitions and templates for those new types.

Having spent a lesson considering data-directed design in isolation, we will now turn to the problem of applying it to lists. In the process, we'll arrive naturally at recursion as a tool for removing the limitations we've been living with so far.

> **Note:** Most of the time in this course, we'll provide you with the data definitions we'd like you to use, so it's more important to be able to read them than write them. Still, it's possible we'll ask you to write down a data definition or a template function in the context of an exam.

Discussion

Topic: Module 05 / 05.2 Data-Directed Design

Hide Discussion

Add a Post

Show all posts | by recent activity

05.2.1 Concept Check — 3
Hi there, I'm wondering why this isn't correct: (define (extnum? val) (or (number? val) (symbol=? val 'infinity))) Is there something about the question that I'm missing?

Q05.2.1: why this is wrong? — 2
(define (extnum? val) (cond [(empty? val) true] [(or (number? (first val)) (symbol=? 'infinity (first val))) (extnum? (rest val))] [else false]))