Lecture 04: User-defined procedures, Design Recipe

10:00 AM, Sep 11, 2019

Contents

1 Sets and Set Notation

2 Functions

3 User-Defined Procedures

4 A Recipe for Design
   4.1 The Design Recipe for Atomic Data
   4.2 Example

5 Summary

1 Sets and Set Notation

One way to describe a set is called restriction: we write something like

\[ U = \{ x \in S \mid 0 < x < 6 \}. \]

That’s read aloud as “\( U \) is the set containing each element \( x \) of \( S \) with the property that \( x \) is between 0 and 6.”

A special set is the empty set, which has no elements. It is represented by \( \{ \} \) or \( \emptyset \). That means for any \( x \), \( x \in \emptyset \) is false. This set comes up often in mathematics, as does its computational analogue in CS.

2 Functions

Our first language is Racket, and it’s called a functional programming language. This use of the word “functional” is not the commonplace one where it means “not broken”, as in “is your car functional, or is the battery still dead?” Instead, it means “based on the mathematical idea of functions.” So let’s talk about those for a few more minutes.

You’ve all probably encountered functions in your algebra class, something like

\[ f(x) = x + 3. \]

Everyone agrees that this denotes a function called \( f \) that adds 3 to a number. But any working mathematician, on being shown this, will probably not say that it describes a function at all. The
mathematician will insist that you specify a bit more: exactly what can \( x \) be? And what kinds of results can be produced? So a mathematician, describing that same function, will write this:

\[
f : \mathbb{N} \to \mathbb{N} : x \mapsto x + 3.
\]

Let us think of functions as machines, that take an input, apply a rule to it, and produce an output. Here \( \mathbb{N} \) is a standard symbol from math and denotes the set of natural numbers, \( \{0, 1, 2, \ldots\} \), and we can read this, from left to right, as saying “\( f \) is a function that takes in natural numbers and produces natural numbers; for a typical natural number \( x \), the value produced by \( f \) is \( x + 3 \).”

The first \( \mathbb{N} \) is the set of items that can be fed into the “function machine”, called the domain, the second is the set of items that are produced by the “function machine” called the codomain, and the thing with the \( \mapsto \) arrow is called the “rule” which is “add 3” here. We say that the function \( f \) consumes things in the domain and produces things in the codomain.

Be careful to not confuse the terms codomain and range. They mean different things in mathematics and cannot be used interchangeably.

This might all seem pretty pedantic — after all, we’re just adding 3 to a number! — but I want briefly to come out in favor of pedantry in this situation. Being really precise about what something means is at the very center of what we’ll do in CS17, and since functions are one of the main objects of study in computer science, getting this right really matters. So almost every time I discuss a (mathematical) function, I’ll use the notation above: I’ll indicate the name, the domain, the codomain, and the rule. The arrow between domain and codomain will be an ordinary one; the one indicating the rule will be a \( \mapsto \) arrow as follows:

\[
\text{Domain} \to \text{Codomain} : \text{var} \mapsto \text{expression involving var}
\]

Not every function we’ll want to look at has a rule that can be written using simple algebraic notation. We’ll see an example or two next time. For instance, we might have a function \( h \) whose domain and codomain are both \( \mathbb{R} \), the set of real numbers, and whose rule is that if \( x \) is negative, then \( h(x) = -1 \); if \( x \) is positive, then \( h(x) = 1 \), and if \( x = 0 \), then \( h(x) = 0 \). For such functions, we can still write something that looks a lot like the formal notation used earlier. We write:

\[
h : \mathbb{R} \to \mathbb{R} : x \mapsto \begin{cases} 
  x & \text{if } x > 0 \\
  -x & \text{if } x < 0 \\
  0 & \text{if } x = 0 
\end{cases}.
\]

This is sometimes called case notation, because it divides things into various cases.

We can even have functions that are described entirely in words, like the function that assigns to each U.S. president from 1960 to 2011 that president’s party affiliation. It turns out that this last category of functions really dominates: many of the programs you write will be ones that compute functions that are best expressed in ordinary words. But the ordinary words used to express them have to be written very carefully to make the function description unambiguous.

### 3 User-Defined Procedures

Last class, we discussed how you can define your own procedures, by writing something like this:
or more generally:

\[
\text{(define (next-number n)} \ (+ \ n \ 1))
\]

We'll name the parts of such an expression like this:

\[
\text{(define (name arg1 \ ... \ argn) body)}
\]

where name and all the args are identifiers, and body is an expression.

Note the ( before name: this parenthesis is the marker for a user-defined procedure— it’s how you know it’s not a user-defined identifier.

4 A Recipe for Design

Today I am going to present a recipe that you will use in CS17 to guide you in the design of simple programs that operate on atomic data (nums, booleans, and strings). Next time, I will add additional steps to the recipe; these will aid you in the design of more complex programs—programs that operate on compound data. Following this design recipe is not “Racket law,” but rather “house rules” in CS17.

While this recipe is valid regardless of programming language, it is applicable only after a programming language has been selected. After learning several languages, you will find that some are better suited to certain tasks than others, and that it is necessary when first presented with a problem to start by selecting an appropriate language. In CS17, however, we will always make the choice of language for you.

Many of the steps in the design recipe are mechanical. This is intended to eliminate the “blank page” phenomenon, whereby students do not know where to begin.

4.1 The Design Recipe for Atomic Data

1. **Data Definition**: say what kinds of data you’ll be working with. Choices are num, bool, and string. [This rule will soon change slightly.]

2. Provide examples of the data the procedure will process.

3. Specify the procedure’s type signature, which describes the (atomic) type of data the procedure consumes, and the (atomic) type it produces.

---

1 In Racket, the num type includes both integers and floating point numbers.
2 In CS17, we pretend strings are atomic.
3 The CS17 design recipes are adaptations of similar recipes that appear in a textbook by Professor Krishnamurthi (and others): *How to Design Programs*—[www.htdp.org](http://www.htdp.org).
4 Often, this requires that you learn a new language!
4. Following the type signature, describe the procedure’s **call structure**, i.e., define the procedure in Racket and give names to the procedure and its arguments, but don’t fill in anything else yet.

5. Write an **input-output specification** for the procedure. That is, *in words*, not code, state the relationship between the procedure’s input and output. As you describe each input argument, you may also possibly constrain the domain of those arguments (e.g., ‘positive integers’ instead of ‘all integers’).

6. Provide **test cases** that exemplify the procedure’s operation. These tests must follow its call structure and satisfy its specification.

7+8. **Code** the procedure. *This step may require you to be inventive, and should be fun, rather than rote*.

9. **Run** your program on your test cases.

---

Old MacDonald wants to know how many eggs he will produce given n number of chickens. Each chicken produces three eggs. Write a procedure, `egg-calc`, that when inputted a number, n, of hens, outputs the number of eggs that old MacDonald will have. **USE THE DESIGN RECIPE**

**Question:** Why write test cases before you write the code for the procedure?  
**Answer:** It makes sure that you understand the procedure before you write the code for it. In general, when you’re writing a function, you should know what the output would be for a given input. Writing test cases makes you think about how you want your function to work, which solidifies your understanding of it.

4.2 Example

Prof. Hughes did a complete worked example with the “count-posts” procedure, starting at slide 15 in the powerpoint slides.

5 Summary

You’ve learned about one of several ways to describe sets (namely, restriction). You’ve learned about domain, codomain, and rules. You’ve also been introduced to the design recipe (or at least one instance of it), a pattern that you’ll be following over and over throughout the whole course. Part of that recipe was a standard format in which to write a “type signature” which tells what sort of data comes into and out of a procedure.

And, if you look at the slides for this lecture, you’ve seen this recipe applied to write and test a post-counting procedure.

---

5 In the next few lectures, this step will be broken down into two steps.
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: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).