Lecture 02: Functions and Racket [PROVISIONAL]

10:00 AM, Sep 8, 2017

Contents

Objectives

By the end of this lecture, you will:

- understand set notation and equality
- be able to identify and describe functions, including "case" notation
- understand how a program is processed using a "REPL" model
- understand the difference between syntax and semantics
- describe a basic subset of the Racket language using Backus-Naur Form
- be able to write a few more small Racket programs

1 Sets and Set Notation

As we discussed in the last class, a *set* is a collection of things.

That's a pretty informal description, but it turns out that trying to formalize the notion of a set is essentially hopeless. So we'll just use this intuitive description,and describe some of the ways we can work with sets.

We can describe sets in multiple ways.

The first is plain English: we say that U is the set of all positive even numbers, for instance, or that V is the set of all students who are taking CS17 this semester.

THe second way to describe a set is via *enumeration* — listing things that are in the set between curly braces — and the set so described contains *exactly* those things. So the set $S = \{1, 5, 2\}$ has 1, 5 and 2 as *elements* (an "element" of S is something that's in S), but nothing else is an element of S. We write $1 \in S, 2 \in S, 5 \in S$.

The third way (which we won't use a lot) is *restriction*: we write something like

$$U = \{x \in \mathbf{Z} | 0 < x < 5\}.$$

That's read aloud as "U is the set of each element x of $\mathbf{Z}$ (the integers) with the property that x is between 0 and 5." We could therefore also have written $U = \{1, 2, 3, 4\}$ to specify the same set.

1.1 Set Equality

Now that we know at least one way to form sets, we can explore the notion of set equality.

For two sets to be equal, they need to have the same elements. Order does not matter. Nor does the number of times the same element appears in a set. In other words, for two sets to be equal, all elements of one set must be in the other set and vice versa.

Some might think that the shorter notation (in which we never list an element twice) means "sets don't contain things twice," but really all the set-construction notation tells you is which statements about "element of" are true.

For example, the sets $\{6, 1, 1, 1\}$ and $\{1, 6\}$ are equal because they both contain 6 and 1; it does not matter that the order is different and that the first set-description contains 1 multiple times.

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:

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

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 called the *domain*, the second is called the *codomain*, and the thing with the $\mapsto$ arrow is called the "rule". We say that the function f *consumes* things in the domain and *produces* things in the codomain.

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.

I want to tell you what a mathematician means by saying that two functions are *equal*. To be equal functions, f and g must

- have the same domain
- have the same codomain, and
- produce the same results, i.e., for every x in the domain, we must have $f(x) = g(x)$.

That means that the function that takes a real number and squares it is *not* the same as the function that takes an integer and squares it, even though both might have been written, in your algebra book, as

$$f(x) = x^2.$$

This might seem crazy to you. You're thinking "You've got the formula – x^2 — isn't that *enough*?" And the answer is "No, not really." Lots of stuff in math seem crazy at first, like the rule that the product of two negative numbers is positive, but after a while, it not only seems natural, but begins to seem like the only way things *could* be. I want to be clear here, though: the choice of defining a "function" in mathematics as having a domain, codomain, and rule, is a purely human one. Mathematicians tried other definitions, and found that it was harder to write clear and unambiguous proofs with those, and eventually, after a few hundred years, settled on this meaning and notation. It's one that works really well, and it turns out to be one that's far better adapted to most of computer science than the "functions are just formulas" version. So it's the one we'll be using.

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 expresssed in ordinary words. But the ordinary words used to express them have to be written very carefully to make the function description unambiguous.

3 A bit more Racket

```
...
(let
   ([alon1   (list 1 2)]
    [alon2   (map (lamda (x) (/ x 4.0)) (list 2 14))])
  (map + alon1 alon2))
...
```

Above is a Racket program. You should be looking at this and thinking "I have no idea what it means". But let's just look at it visually. The "..." means that I am showing you a part of a larger program. There are some parantheses and braces, there are some lists of characters, there are some things that look like numbers, there is a slash...

I said "no magic," but this is the first place where I won't rely on magic, but on something that you probably believe: it's possible to write a computer program that looks at text and separates it into parts in some way. I'm suggesting this based on your experience. You've probably used "Find" in your favorite word processor and it manages to find a particular piece of text. This "finding" process is a lot like being able to find parentheses, or numbers, or slashes, etc. Anyhow, believing that it's possible to break program text into those fundamental pieces is the first thing I'm hoping you'll believe without proof, for at least a couple of weeks.

While I'm getting you to commit to things, I am also going to get you to commit to the idea that somehow, inside the box, a computer can do arithmetic.

OK, with those two commitments, let me tell you a little more about how Racket programs get processed.

As I said already, the first step in processing Racket programs that we will be writing is going to be that we look at them and break them into parts:

- Whitespace: blanks, tabs, line breaks, etc
- Punctuation: for now, only parentheses and square braces
- Other stuff: More on this as we go along.

And I'm going to assume that that process of taking program text and turning it into separate bits is doable. I am going to build up from those recognizable bits to whole programs. And to say what's a "legal" program, I am going to describe the "shape" of program pieces.

We are going to start with a very limited version of Racket. The entities that we are going to be working with are numbers. Of course, they will be more entities later, but for now numbers are going to be the only allowable entities. Numbers are things that look like a bunch of digits, maybe a decimal point, maybe a "+" or a "-" sign in front. And to that, I'm going to add a typewritten version of scientific notation, one in which a number like 1.7×10^6 is written `1.7E6`. Negative exponents are also allowed, like `1.7E-6`, and for symmetry, you can use a "+" sign in there as well: `2.3E+12`. But no whitespace is allowed within a single number, so `2.3 E +12` is *not* an allowable written form for a number.

In our model, the processing of a Racket program can be broken down as a formulaic set of three main steps:

- **Reading.** The first step in processing a program involves reading the program text. Doing so involves something known as *tokenizing*: breaking the code down into individual components such as parentheses, words, etc. Next, *parsing* involves recognizing program entities and somehow representing them in the computer; luckily, this is DrRacket's job and not yours.
- **Evaluating.** Although this step should really be "processing-or-evaluating," it has historically been referred to as simply "evaluating." This step involves taking the entities resulting from the *reading* step and producing something new. As an example of non-racket evaluation, in arithmetic we say that "$17 + 18$ evaluates to 35".
- **Printing.** The last step involves displaying the resulting value, if appropriate. The printed representation of things is usually very similar to what is originally typed into the program.

This structured Read-Evaluate-Print set of steps can occur numerous times throughout the process of evaluating a program. Due to its cyclical nature, this set of steps is referred to as a "Read-Eval-Print Loop," or a "REPL" for short.

I'm being a little sloppy here: I've used the words *program*, *text*, *entity*, *value*, and probably some others here without clear definitions. I'll come back to each of these.

Equipped with a basic model for processing Racket programs, we can begin to examine the components of the Racket language. We will start by describing a small subset of the language. To do so, I am going to use a certain notational form that is really useful: the "Backus Naur Form," or BNF. I won't give rules for BNF, because it's really meant to just be a convenient and compact way for you to remember what's OK and what's not, and since *you* won't ever have to write any BNF, spelling out the rules would be pointless. I think you'll have no trouble picking it up.

```
BNF:
<program> ::= <expression>
<expression> ::= <number>
```

What this means is that in our current subset of the Racket language, a Racket *program* is composed of an expression. The"::=" can be thought of as meaning "is defined as", and things in angle-brackets are just names. So the first line says "something we're going to call a *program* is just a different name for something we're going to call an *expression*." The second line says that an expression is just a number, and not written out at all is the idea that a *number* is one of those things that we said we could identify during the "reading" process: it's a one or more digits, along with perhaps a plus or minus sign, amybe a decimal point, etc., but no white-space.

What all this means is that we can recognize that a program text like

```
44
```

is actually a valid Racket program, because 44 is a number, which is an expression, which is a program.

You might be asking "Why did you need the idea of an 'expression' here at all? Couldn't you have said a program is a number?" Yes, I could. But in just a few moments, I'll be extending the language a bit, and I was anticipating that.

3.1 Syntax

The BNF for Racket tells us what things constitute legal Racket programs, and right now, there aren't very many of those, and they're not very interesting. Soon the collection of legal programs will grow much larger. But these rules are just rules saying how things can be put together, not what they *mean*. In English, we have similar rules: we say that a sentence must begin with a capital letter and end with one of several punctuation marks, and follow various other rules as well, so that `Bread is good.` is a syntactically valid English sentence, which happens to also make sense to us, and with whose meaning many of us agree. On the other hand, `sLate ; hoT carry` is not syntactically valid (except in some wilder forms of poetry, perhaps). The component parts — punctuation, words, capitalization — are in the wrong places.

3.2 Semantics

There's another characteristic of languages, called *semantics*, which has to do not with what is legal in structure, but what is actually meant. For Racket, the semantics will be defined by certain "rules of processing." These tell you what you can expect to happen when you run your program.

As our known subset of Racket grows, so will the rules of processing. There will be two main rules of processing (one for expressions, which you're about to see, and one for definitions, which we haven't yet discussed).

The rule for expressions is messy enough that it's broken out into several pieces, called the "rules of evaluation." Since these really completely describe what your programs do, at some point you'll want to memorize them. You might want to wait a couple of days, though, until they're finalized.

(Preliminary) Rules of processing: An expression is processed by evaluating it.

N.B. "Evaluating" is an undefined term now, but I will describe the preliminary rules of evaluation in just a moment.

The result of evaluating an expression is a *value* (that's a new word. A quick gloss: a value is the result of the as-yet undefinded evaluation operation).

When you have an expression that is a program, the *printed representation* [1] of the value gets printed out.

(Preliminary) Rules of evaluation: The value of an expression, which has to be a number, is that number.

This is a terrible thing to write, because it's using the word "number" in different ways. I'll explain that in a moment. But for now:

As a result, when we type a number into DrRacket, we get back the same number.

Just to explain that last sentence in more detail, let's look at it step by step.

A lot happened behind the scenes. Racket took the text "44", made some *internal representation* of it, looked at that and said "Aha! That is a program, because it is an expression, and an expression is a number and that is what I've got." And because it was a syntactically legal program (it "matched up" with the BNF description of Racket), Racket proceeded to invoke the rules of processing on

[1] Another new word. For now: the printed representation of a number is a sequence of characters usually used to denote that number in day-to-day discourse.

the program. The rules of processing say that Racket must process an expression by evaluating it. So Racket says to itself, "Oh, my expression is a number expression! That means (by the Rules of Evaluation) that its value is the same number!" So Racket created a separate thing, a number-*value*, which also represents the number 44, but perhaps differently.

And then because that expression constituted an entire program, the printed representation of that value was printed out for us, and we saw the characters `44`.

If this behavior was all that we wanted from a program, breaking things into read-eval-print loops, and describing them with BNF, and distinguishing between input text and values and printed representations would be pretty silly. We could just describe the whole process by "type out whatever the user types in." But we actually want something better than that.

So now let's move on to a slightly richer language.

4 Summary

Skills

- For two functions to be equal, they must have the same domain, the same codomain, and the same rule.
- For two ordered pairs (a, b) and (c, d) to be equal, a must equal c, and b must equal d.
- Sets are a collection of things. We will call these things elements. We use the symbol $\in$ to represent that an element is in a set. Order in a set does not matter; duplicates in a set do not matter.

Ideas

- Functions have a domain, codomain, and rule.
- The Cartesian product of two sets is the collection of all ordered pairs in which the first element comes from the first set, and the second element comes from the second set.
- The processing of Racket programs can be broken down into three steps. We'll call this structure Real-Evaluate-Print-Loop, or REPL, due to its cyclical nature.
  - Reading - we break the program text down into tokens through a process calld tokenizing. Then, DrRacket parses the programs.
  - Evaluating - We take what we've found from the reading step, and produce something new. For example, we'll evaluate $17 + 18$ to be 35.
  - Printing - we will display the resulting value, when appropriate.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/cs017/feedback`.