Lecture 02: Functions and Racket

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, describe and check for equality of functions, including "case" notation
- understand how a program is processed using a "REPL" model
- be able to break down a program using tokens
- understand the meaning of syntax and semantics
- describe a basic subset of the Racket language using Backus-Naur Form

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 $S = \{1, 5, 7\}$ has 1, 5 and 7 as *elements* (an "element" of S is something that's in S), but nothing else is an element of S. The statements $1 \in S, 7 \in S, 5 \in S$ are all true, but $x \in S$ is false unless $x = 1, 5, 7$.

The third way (which we won't use a lot) is *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.

1.1 Set Equality

Now that we know at least one way to form sets, we can explore the notion of set equality. Think about whether the sets $\{1, 2\}$ and $\{2, 1\}$ are equal.

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:

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

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:

$$Domain \to Codomain : var \mapsto expression\, involving\, var$$

2.1 Function equality

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 absurd 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 absurd 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. Furthermore, the domain and codomain cannot be inferred from just looking at the rule, and you are not yet expected to know how to choose these. They will always be specified for you.

Here's why knowing how to test for equality is important. In Computer Science, two functions producing the same output can have very different running times, and one of the goals in CS is to make faster programs. If we have an accurate program that is slow, we can work on it to make it faster and compare it to the original for accuracy.

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 The Read-Eval-Print Loop (REPL)

When a computer processes a Racket program it executes a Read-Eval-Print Loop or REPL. This loop involves input and output are in a form understood by us, i.e., text, and internal representations that the computer has for all the steps in between. To give a real-life analogy, when the number 2 is said aloud, people can have different representations of it in their heads. Some may think of two spelled out, some the numerical symbol, some the roman symbol. The *abstraction* involved here allows many different representations, as it gives little information about how "two" should be thought of. Don't worry too much about what *abstraction* means right now, we will learn about it in detail later. Coming back to the REPL, it involves 3 basic 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 representing them internally 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 a "value". 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.

3.1 Tokens: the building blocks of languages

Syntax is a set of rules that say what's allowed in a language. 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 "`I am here.`" is a syntactically valid English sentence, which happens to also make sense to us, and with whose meaning many of us agree. "`Granite thinks carefully`" is a syntactically correct but weird sentence. On the other hand, "`Frog hunger red.`" is not syntactically valid since there is no verb. Racket has a syntax, and so does CS17 Racket (the parts we'll use) which is in fact only a page long.

The first step to understanding syntax is breaking a program into "tokens". These are building blocks, like words or punctuation in English. 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. 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. CS17 Racket tokens are a few punctuation marks (for now only parenthesis and square braces) a few "keywords," numbers, and names, and are separated by whitespace. You can read about the rest in the Racket Documentation but this is not necessary for the course.

Let's identify the tokens in the following example:

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

The "..." means that I am showing you a part of a larger program and are not Racket tokens. The parenthesis and square brackets are punctuation, `let` is a keyword, `alon1`, `/` and `+` are names, and `4.0` and `14` are numbers.

Describing tokens in typically done very formally in courses on Programming Languages, but we will skip this formality for numbers and names as these rules are Racket-specific and messy. In Racket, the different tokens you will encounter are:

- **Punctuation**: `( )[ ] ;''` serve as separators, like whitespace, helping to separate tokens. Other like `# ' ` | { }` will never come up in CS17.
- **Numbers**: 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.

- **Keywords**: `let` is a keyword which we saw earlier. Others will be introduced as we go. For now, forget you've ever seen `let`
- **Names**: Nonempty sequences of non-special, non-whitespace characters, starting with non-digit (this is a CS17 rule), and which cannot be understood as numbers or keywords are names. Examples are `book`, `table1`, `+`, `my-long-and-silly-name`. As a note, in CS17 using `-` to separate multi-word names is preferred and using underscore (i.e. `_`) is disallowed. Additionally, use informative names and stick to lower case. Non-examples would be `13book` because it starts with a digit and `blo(b` because it contains a special character.
- **Strings**: A sequence of any characters (except for double quotes), between double quotes. Examples are `"this is a string"`, `"this (really)is a string too"`, `""` which is the empty string, `"17"` which is unrelated to the number `17`.

Tokens are separated by **Whitespace**: blanks, tabs, line breaks, etc. Note: use only upright double quotes for strings, and **no smart quotes** as they will lead to errors. If you only work in DrRacket you will never produce these!

3.2 From tokens to language

To "define" a programming language, we specify which token sequences are allowed. The rules of what is allowed, as we have seen, is called syntax and will be disclosed gradually over the next couple weeks. Syntax will describe how to write in the language more generally. We'll also assign *meaning* to each allowable token sequence with a set of rules, which is called *semantics*. For Racket, the semantics will be defined by certain "rules of processing" and "rules of evaluation". 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. 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 until we formally finalize them though.

Equipped with a basic model for processing Racket programs, we can begin to examine the components of the Racket language. 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 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, because a formal description will be very messy, 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, maybe a decimal point, etc., but no white-space.

Coming back to the Rules of Processing, it is too hard to define exactly what a program **means**. Additionally, we don't currently know what happens when we **run** a program as it contains many parts intertwined, but we do have a model of the Read-Eval-Print-Loop. We will explore this idea in detail in the coming lectures.

4 Summary

- 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.
- Functions have a domain, codomain, and rule. For two functions to be equal, they must have the same domain, the same codomain, and the same rule.
- 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 called 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.
- A program can be broken down into tokens, which are the building blocks of the language.
- To "define" a programming language, we specify what token sequences are allowed and what meaning they carry. For this we use a notational form called BNF.

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`.