# Lecture 02: Functions and Racket

*10:00 AM, Sep 6, 2019*

## Contents

**1 Announcements** **1**

**2 Review** **1**

2.1 Tokens: the building blocks of languages . . . . . 2

2.2 From tokens to language . . . . . 3

**3 Summary** **4**

## Objectives

By the end of this lecture, you will:

- know how to recognize a legal Racket program, at least for the small part of Racket we've encountered so far
- know about three ways to describe sets
- know the names $N$ and $R$ used to denote the natural numbers and real numbers
- know about a new data type, boolean.

## 1 Announcements

- Register ASAP, so that we can get you into a lab section for Sunday and create an account on the CS department machines. This is very important, since it is the only way that you can get graded. Failing to do so will result in you not being assigned a lab time this week, and consequently a grade of 0 on your first lab assignment.
- If you have not received an email regarding lab signups and homework release, contact the HTAs. Their email address is: `cs017headtasbrown.edu` .
- Homework about collaboration policy, some BNF things and Racket arithmetic expressions will come out later today. You will get to know the collaboration policy better through the homework about the important message is don't take notes away from discussions about assignments. We check people's homework for similarities. If two programs are character for character then I will make an assumption about it. There is also a challenge problem that involves challenges somewhat ahead of what I expect students doing this for the first time to know.

# 2 Review

Recall that we've seen two simple kinds of programs: simple "expressions" consisting of a single number (or some slightly more complex things like `(+ 3 2)`), and definitions, which looked like `(define height 12)`. We said that a CS17 Racket program consisted of one or more definitions, followed by zero or one expressions.

Today, we're going to try to make that a little more precise. There are very specific rules defining what is a program and what is not. Take the programs we wrote so far and match them to various pieces. The program `17` is an expression, a number expression, and it is actually a number `(define height 36)` is a definition because it starts with the word define. It has the keyword define and the name of the thing height and the expression, number expression, and number,36. Height is also an expression, a name expression and is also a name. `(+ 3 5)` is another expression. 3 and 5 are both number expressions and numbers, + not defined for us yet. The word "is" is tricky, it can be explained by the fact that there are lots of things that I am just like the `17` is multiple things.

To get even more precise, we start by breaking the problem ("How do I describe programs?") into smaller pieces. we'll first describe the most basic program pieces ("tokens") and then describe the ways we're allowed to assemble tokens into programs. The language has rules just like English has rules. A basic rule is that the parentheses always match up in pairs so we gain a basic level of understanding of the program just by looking at the characters of text without trying to describe meaning to them. This is like asking if something is an English sentence which needs a subject and predicate. Ran far has no subject, so it sounds weird. Goal in this class is say the word so you understand stuff and then give a simpler definition. Here is the structure of a definition: `(define <name> <number>)` you don't know the rules for numbers yet but any list of digits works. Rules for a name is a sequence of letter numbers and dashes. That doesn't start with a digit and doesn't contain any other punctuation. Underscores aren't allowed but dashes are. Sentence construction -what is okay to type. I'm not going to sweat what it might mean. First part is syntax,which is made up of tokens, the second part is semantics-meaning- which we will ignore for now.

## 2.1 Tokens: the building blocks of languages

Tokens are a few punctuation marks (parentheses, square braces, double-quotes) a few "keywords," numbers, and names, and are separated by whitespace.

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. Others 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 $1.7 \times 10^6$ 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**: `define` is a keyword which we saw earlier. Others will be introduced as we go.
- **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`.

Identify the tokens in these racket programs, for each one say what type it is: `22e4 (+3 7.2) (name1 name2 name3)`

Now that you've tokenzie some Racket programs, broken them up into bits.

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.

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!

## 2.2 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.

## 3 Summary

With the main ideas of sets complete, we're now ready to move on and define *functions*, and their computationalanalog, *procedures*.

- 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 CS 17 document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/csci0170/feedback`.