(Provisional) Lecture 03: Definitions, Procedures, Environments [PROVISIONAL]

10:00 AM, Sep 11, 2017

Contents

Objectives

By the end of this class, you will know:

- How strings, definitions, and names generally work in the Racket language
- The basics of bindings and environments
- Some built-in Racket procedures

1 The Racket Language (Version 1)

Remember the BNF we saw in the last class:

```
BNF:
<program> ::= <top-level-expr>
<top-level-expr> ::= <expr>
<expr> ::= <number>
<number> ::= messy stuff I'm not writing out
```

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” or “consists of”, and things in angle-brackets are just names. Such things are called *entities* in BNF. 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, maybe 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.

1.1 Syntax

BNF defines the *syntax* of the Racket language: the things you’re allowed to write and have someone call is Racket. In English, you can write “I like to eat kale” or “I like to eat malaise”; they’re both syntactically correct sentences. Only the first makes any sense (and those who don’t like kale might quibble even with that). But *sense* is outside the domain of syntax. What you *can’t* write is “Furious hit ran.” There’s no subject, two different verbs, and the transitive verb “hit” lacks a direct object. *That’s* a syntax problem.

1.2 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. "Expression" here refers to the part of a program that matches `<expression>` in the BNF.

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 "number rule": The value of an expression, which has to be a number, and corresponds to some ordinary number x, is the number-value that represents x.

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 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! That means (by the Rules of Evaluation) that its value is the number-value that represents 44. And the printed representation of that number value is the characters "4" and "4". 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.

2 More Racket

We've got a version of Racket that looks like this:

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

2.1 Racket (Version 2)

Now let's introduce **strings** to the mix. A string is basically a sequence of characters – details in just a moment. We consider strings to also be expressions, so we write this:

[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.

```
BNF:
<program> ::= [<top-level-expr>]
<top-level-expr> ::= <expr>
<expr> ::= <number> | <string>
<number> ::= messy stuff I'm not writing out
<string> ::= stuff between double quotes
```

The vertical bar is part of BNF, and is read "or". So now an expression is either a number *or* a string. The square brackets around $\langle$*top-level-expr*$\rangle$ is also part of BNF and means "zero or one of these," or in other words, "optional". Of course, programs that contain nothing at all don't come up very often.

So what's a string? A **string** is a double-quote–delimited collection of characters that does not include double-quotes.

> **Question:** Which of the following are valid (i.e., "syntactically correct", i.e., matching the BNF) Rackette programs as of now?
>
> - `17`
> - `17 18`
> - `"17 18"`
> - `"This is not a program"`
> - `"This is "my"program"`
>
> **Answer:** The first, third, and fourth are syntactically correct. The second is not, because it consists of two numbers, while our BNF only allows one. The fifth is not because there are quotation marks within the string (or, more accurately, because it consists of a string `"This is "`, followed by the two letters `my`, followed by the string `"program"`, and that's not syntactically valid.

2.2 Revised Rules of Processing

Given our new BNF, we must now revise our rules of processing and evaluation.

Rules of processing v. 2: An expression is processed by evaluating it to produce a value v.

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

Rules of evaluation v. 2:

1. The value of an expression that is a <number>, representing some ordinary number x, is a number-value representing x. Short form: "The value of a number is that number."

2. The value of an expression that is a `<string>`, representing some ordinary text T, is a string-value representing T. Short form: "The value of a string is that string."

I promise it gets better from here.

3 Definitions (Version 1)

So far what we've got is a language in which there are numbers and strings, and these things have "meanings" that are pretty much what we expect.

Certain numbers end up being used often in programs. If you were writing a program to work with English measurement units, you might want to use the number 5280, the number of feet in a mile, fairly often. But it's possible that this number might arise in some other context in your program. Replacing this number, 5280, with a name that communicates the exact use of this number — a value that converts miles to feet — might be a nice thing to do. Racket lets you do that:

```
(define feet-per-mile 5280)
```

Once you've made that definition, you get to use `feet-per-mile` in the remainder of your program instead of 5280.

Three remarks:

- `feet-per-mile` involves more typing, but it also involves more clarity. You'll soon be glad to have that degree of clarity in your program.
- I've just introduced new syntax and the *reason* for that new syntax before introducing the BNF or the rules of processing for that new syntax. That's a pattern I'll follow from now on.
- The name I chose to use — `feet-per-mile` — might look like something involving two subtractions. it's not, but that's because the rules of Racket are different from the rules of traditional algebraic notation.

3.1 Syntax

The revised BNF to allow for definitions is this:

```
BNF:
        <program> ::= <defn>* [<top-level-expr>]
        <defn> ::= <name-defn>
        <name-defn> ::= ( define <name> <expr> )
        <top-level-expr> ::= <expr>
        <expr> ::= <number> | <string>
        <number> ::= messy stuff I'm not writing out
        <string> ::= stuff between double quotes
        <name> ::= sequence of non-special, non-whitespace
           characters that's not a number or a keyword
```

Again, the square brackets surrounding `<top-level-expr>` represent "zero or one of these". The asterisk next to ⟨*defn*⟩ represents "zero or more of these". So a program now looks like "a sequence of zero or more definitions, followed by zero or one expressions."

We see that a **definition** must begin with a literal open-parenthesis, the characters d-e-f-i-n-e in order, with no spaces between them, and then an identifier followed by an expression and a close-parenthesis.

Note that we will call the expression on line 1 a *top level expression*, however, the expression nested within parentheses on line 4 is *not* a top level expression. We have mainly been looking at top level expressions so far, but will go into more depth with these nested expressions soon.

An *name* (or *identifier* — both words get used), like a number or a string, is something that is recognized during the "read" part of the REPL: it's a sequence of non-whitespace, non-special characters (more on this in a moment) that is not a number or a keyword. We'll slightly refine this description in a few days.

3.2 Semantics

To describe the semantics of bindings, I need to remind you of something I said you'd have to accept back on day 1: A computer program can associate one thing with another.

As an example, your operating system associates a filename with the contents of that file. In the case of names, we'll be associating "values" to names. We'll say that the name is bound to the value. And we'll arrange that this "binding" of names to values is a *function* in the mathematical sense: any name is bound to one value, not to two or three or twelve. (To be more precise: most names are unbound, but any name that *is* bound will be bound to exactly one value.)

So a "binding" is an association of a value to an name. We'll say that a collection of bindings is an **environment**, and for now, there's just one, which we'll call the **top-level environment**.

I like to think of this as a piece of paper on which I write names on the left and values on the right. The environment is the piece of paper; a binding is one line on that page. (This is an example of a *model*. Clearly, inside my computer there is no actual paper or person writing things, etc., but it's good enough to give me predictable results.)

That piece of paper serves sort of like a dictionary, and we can think of "looking something up" as finding the name in the left column and responding with the corresponding value from the right column.

With all those ideas out of the way, I can now enlarge the rules of processing and evaluation.

Rules of processing v. 3: An expression is processed by evaluating it.

When you have an exression that is the sole expression following a sequence of definitions, the *printed representation* of the value gets printed out.

A definition

```
(define <ident> <expression>)
```

is processed by checking that the name is not bound in the top-level environment. (If it's already bound, that's an error, and processing stops.)

If the name is not bound, then the expression is evaluated to produce a value v, and a new binding is placed in the top-level environment binding the name to the value v.

Rules of evaluation v. 3:

1. The number rule (see above).
2. The string rule (see above).
3. The value of a name is the value to which it is bound in the top-level environment, or, if there is no such binding, an error occurs and evaluation and processing stop.

As such, we now have the following revised BNF:

```
        BNF:
                <program> ::= <defn>* [<top-level-expr>]
                <defn> ::= <name-defn>
                <name-defn> ::= ( define <name> <expr> )
                <top-level-expr> ::= <expr>
                <expr> ::= <number> | <string> | <name>
                <number> ::= messy stuff I'm not writing out
                <string> ::= stuff between double quotes
                <name> ::= sequence of non-special, non-whitespace
                   characters that's not a number or a keyword
```

We can now write programs like this one:

```
(define feet-per-mile  5280)
feet-per-mile
```

The processing of this program takes several steps: first, the definition is processed, resulting in the binding of `feet-per-mile` to the value 5280.

Then the top-level expression `feet-per-mile` is evaluated using part 3 of the rules of evaluation, resulting in the value 5280. Because it's a top-level expression, the printed representation of that value gets printed out.

> **Question:** Suppose we process the following program:
>
> ```
> minutes-per-hour
> ```
>
> What happens?
>
> **Answer:** We end up at Rule 3 of the rules of evaluation, and because the name `minutes-per-hour` has not been defined, we get an error.

4 The top-level environment

You might be asking yourself "How do we know that `minutes-per-hour` wasn't already defined in the top-level environment? Does it start out empty?" The answer is "No, it doesn't start out empty, but the things that are in it have unusual enough names that you're not likely to stumble on any of them." So you couldn't *know* that this name was undefined, but you could guess it.

But it turns out that several very important things *are* predefined, and have names that come up pretty often. One of these is +, and if you ask DrRacket what its bound to, you'll get this answer:

```
> +
#<procedure:+>
```

which suggests that the plus-sign is bound to something rather different from anything we've yet encountered.

We have common names for concepts like "number" and "string of characters", but we don't have a common name for the thing that + is bound to. So talking about it will take a little work, which I want to delay for a little while and talk about a new kind of expression

5 More Racket: the good stuff!

I'm now going to describe *procedure-application expressions*, which look like

```
( <expr>  <expr>  ... <expr> )
```

i.e., they look like a sequence of expressions in parentheses. There can be any number, one or more, of expressions between the parens.

It's helpful to give these names, and write the expression in the form

```
( proc arg1 arg2 ... argn )
```

The rule for evaluating this kind of expression, which follows all the prior rules, is then

1. Evaluate `proc`; if the resulting value is not a procedure, it's an error and evaluation stops.
2. If the resulting value is a procedure p, then evaluate `arg1` ... `argn` to get values $v_1, \ldots, v_n$.
3. Apply the procedure p to the values $v_1, \ldots, v_n$ to get a value v; v is the result of evaluating the procedure-application expression.

Now that last step — "Apply the procedure ..." — is pretty vague. But in the case of something like addition of numbers, the meaning is familiar to you: applying addition to the numbers 2 and 17 results in the number 19. Racket's addition procedure can take any number of arguments, even zero, and the sum of no numbers at all is chosen to be zero (i.e., Racket's designers thought that was a good choice).

Without going into details, we now have a language in which we can do arithmetic, since I'm telling you right now that addition, subtraction, multiplication, and division are all defined in Racket, with the commonplace symbols for these operations as the names that are bound to the corresponding procedures.

Let's finish up by actually evaluating one procedure-application expression. Consider

```
(+ 3 14)
```

That's a program, because it's an expression that's a procedure-application expression. We can tell, because it starts with an open-parenthesis, but that parenthesis is *not* followed by a keyword, so it cannot be a definition.

To evaluate this, the rules tell us to evaluate the first expression, which is the name +. To evaluate an name, we look to see whether it's bound in the top-level environment, and it is (because DrRacket predefines a bunch of useful things like this). It's bound to the builtin addition procedure.

Because that value is a procedure, the evaluation process continues. We evaluate the second expression, 3, which turns out to be a number, so it evaluates to the number-value 3. And `14` evaluates to the number-value 14. We then apply the addition procedure to the values 3 and 14 to get a resuting value 17, which is the value of the procedure-application expression, whose printed representation then gets printed.

We can do more complicated things, too, like evaluating

```
(+ 3 (* 2 5))
```

The evaluation of the third expression requires some additional work — it's another procedure-application expression! — but in the end, the result is 13.

6 Terminology

So far we have expressions and definitions and names as parts of our program. We have something called "values", which show up as a result of "evaluation". We have "ordinary numbers" — the things you use to do arithmetic every day — and *representations* of numbers. But our representations come in two different types: we have some way of representing number-expressions, and a way of representing number-values.

Evaluation is an operation — something very like a mathematical function — that consumes expressions and produces values. To fully describe it, I have to tell you what expressions are (I've been working on that!) and what values are (I've been pretty vague about that). And I have to tell you the rule for getting from one to the other (I've been better about that).

I'll also sometimes say "the value of *this* is *that*", which is shorthand for "the result of applying the rules of evaluation to *this* expression is *that* value."

7 Booleans

Let's close out the day with one more basic type of expression: booleans. These are meant to indicate the notions of truth and falsehood, and so there are exactly two booleans: **`true`** and **`false`**.

To be more precise: there are two boolean expressions. There are also two boolean *values*, and the value of the expression **true** is the value representing "true", and similarly for "false".

Our BNF gets slightly modified:

```
...
<expression> ::=  <number> | <string> | <ident> | <bool>
<bool> ::= true | false
```

and our rules of evaluation get a new part:

The "boolean rule": The value of the expression **true** is the boolean-value representing 'true'; the value of the expression **false** is the boolean-value representing 'false'.

8 Summary

Ideas

- Racket programs consists of definitions and expressions. Definitions bind identifiers to values. Expressions are (so far) numbers, strings, booleans, and identifiers.
- The collection built-in definitions (such as +), followed by your definitions, constitute the top-level environment.
- A procedure-application expression consists of a sequence of expressions between matching parentheses. During processing of a prrocedure-application-expression, the value of the first expression must be a procedure, which is applied to the values of the remaining expressions to produce the value of the proc-application expression. For example, the expression (+ 1 2 3 4 5 6), when evaluated, produces a value whose printed representation is `21`.

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`.