Lecture 04: More Definitions
10:00 AM, Sep 13, 2017

Contents

1 Announcements 1
2 Evaluating 1
3 User-Defined Procedures 5
4 Summary 6

1 Announcements

- Homework 2 will be released today. Material in this homework will be covered in lecture today and on Friday.

2 Evaluating

Defining a name establishes what's called a “binding” in the top-level environment. These “bindings” are then used to evaluate names we encounter in our racket program: the value of any name is just its associated value in the top-level environment (i.e. the value it’s “bound” to). If there's no binding associated with a name, racket will return an error.

For instance, if we wanted to define the name my-favorite-number in our racket program, we’d write the following definition: (define my-favorite-number 17). As stated above, this definition statement would establish a binding in the top-level environment between the name my-favorite-number and the integer 17. Now, any time we were to refer to my-favorite-number later in our racket program, it would evaluate to the number 17!

Let’s recall the syntax of Racket, so far:

```plaintext
<program> ::= <defn>* [<top-level-expr>]
<defn> ::= <name-defn>
:name-defn ::= (define <name> <expression>)
<top-level-expr> ::= <expression>
<expression> ::= <number> | <string> | <name> | <bool> | <proc-app-expr>
<proc-app-expr> ::= (<expression> <expression>*
```

A number is simply character(s) that look(s) like a number. A string is a sequence of non-double-quote characters, between double-quotes. A boolean is either true or false. A name is a sequence
of non-whitespace, non-special characters that is not a number, string, or keyword. For now, we’ve
learned the `define` keyword. In the next lecture, we will learn a few more!

And the rules of processing? In brief: “To process a top-level-expression, evaluate it, and print out
the printed representation of the resulting value. To process `(define name exp)`, check that `name`
is not yet bound in the TLE (or report an error). Evaluate `exp` to get a value `v`, and then bind
`name` to `v` in the top-level environment.”

**Question:** Why do we differentiate between `[<top-level-exp>]` and `<expression>`?

**Answer:** Top-level-expressions and normal expressions are both evaluated the same way.
However, after the evaluation, top-level-expressions are printed, whereas normal expressions are
not.

Fortunately for us, the rules of processing will never change again. This is the very last version. So
it’s worth memorizing.

The short form of the “define rule” is that defining something makes the name `name` be “replaceable
by the value `v`”. But the rules of processing and rules of evaluation actually make it clear just how
this apparent replacement takes place. So while the intuitive notion is helpful, the explicit rules
should always be your backup.

The rules of evaluation are still evolving. The first three are simple: an expression that is a number
evaluates to a value representing that number; the same goes for expressions that are strings and
expressions that are booleans. The fourth is more subtle:

An expression that is a name `ident` is evaluated by checking whether the name is bound in the
current environment; if not, it’s an error and evaluation and processing halts.

If the name is bound, in the current environment, to a value `v`, then the value of the expression is `v`.

Now implicit in there is a notion of a “current environment”. But definitions place bindings in the
top-level environment. What’s the “current environment”? Answer: During processing, there’s a
current environment which is initially the top-level environment (TLE). Later we’ll see rules in
which it’s something different, but for now, it’s always the TLE.

We can now write a slightly interesting program:

```scheme
(define kilometers-per-mile 1.6)
kilometers-per-mile
```

The processing of this program involves binding a name to a value, and then evaluating an expression
that’s a name, which “looks up” the value in the current environment (the TLE) and gets the value
1.6, which is printed.

Now we’re finally going to make the language a little more interesting, by allowing you to do
arithmetic.

First things first: What’s in the top-level environment when you start up DrRacket? Last lecture, I
drew a blank slate, but let’s experiment. If you make a program whose sole content is the character
“+”, and process it, what happens?

Well, it’s got to be either one or more definitions or an expression. It doesn’t have the right parts to
be a definition (there would have to be a parenthesis, for example!), so if it’s going to be a program,
that sole plus-sign must be an expression. It’s not a number or a string or a boolean, and it’s not a procedure application expression (there would have to be a parenthesis), so it’d have to be a name. And indeed, it’s a name: it’s a sequence of non-whitespace characters that’s not recognizable as a number, string, boolean, or keyword.

The rules of evaluation say that to evaluate a name, we look in the TLE to see whether it’s bound to anything.

And this is the part I didn’t reveal in the last class: it is. It’s bound to a thing called “the built-in addition procedure.” There are a few thousand built-ins in the top-level environment, but you’ll only need to know about 20 of them. The printed representation of a procedure is a little tricky: someone has to make a choice for how to represent something for which ordinary mathematics and writing have no fixed notation. The folks who wrote DrRacket have made various choices over the years, and in the pure racket language, it’s printed as

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

But in the beginning-student language, you’re not even allowed to have expressions whose values are procedures, so it’s an error. And in the advanced student language, which is pretty close to the version of Racket you’ll be implementing, the addition procedure’s printed representation is just a plus-sign.

Despite these variations, the important things here are

- The name + is bound, in the top-level environment, to a value that’s a procedure.
- So are -, *, /
- All of these things that are defined for us initially will be called “built-in procedures” or “builtins”; there are about 20 of them that you’ll end up needing to know.
- We’ll also, in a while, be able to create new procedures ourselves. These will be called “user-defined procedures”. The rules of evaluation will be slightly different for those and for builtins.
- We don’t yet know how to do anything with procedures, whatever they may be.

We’re going to remedy that last item right now.

Procedure-application-expressions have the form:

```racket
(<proc> <arg1> ... <argn> )
```

where each of those is an *expression*. We evaluate such an expression in a 3-step process:

1. Evaluate the first expression; the value must be a procedure, \( p \), or it’s an error.
2. Evaluate each of the remaining expressions, in order, to get values \( v_1, \ldots, v_n \). The remaining expressions are called arguments. We’ll call them args for short.
3. Apply the procedure \( p \) to the values \( v_1, \ldots, v_n \) to get a resulting value, which is the value of the procedure application expression itself.
In the case of built-in procs like addition and multiplication, “apply the procedure” has a meaning that’s already pretty well understood, so I’m going to omit details. (To learn more about how a modern computer actually adds numbers, take CS33!)

To remind you of the three-step process: In a procedure-application-expression that looks like (proc arg1 arg2 ... argn), evaluate proc; evaluate args; apply the resulting procedure to the resulting values. Note: here, a “proc” is an expression (not a procedure) that evaluates to a procedure. Namely, “proc” is bound to a procedure in our environment.

Examples, worked out in detail:

(+ 9 8)

This is a procedure-application-expression because it is composed of a pair of parentheses surrounding a sequence of expressions: +, 9, and 8. The rules for evaluating a procedure-application-expression say “evaluate the first expression”. The first expression is +, which is a name. The evaluation rules for names say to look them up in the top-level environment. There, we find + is bound to the built-in addition procedure.

The rules for evaluating a procedure-application-expression also say to evaluate the following expressions, in order. They are both numbers, which evaluate to themselves, so we have the values 9 and 8.

The last part of the procedure-application-expression rule says to apply the built-in function to these arguments, and get the sum of 9 and 8, which is 17.

So we write:

(+ 9 8)
=> 17

The little => is an informal CS17 notation for “evaluates to”.

Brief exercise: What “entities” in the BNF for Racket, does (+ 9 8) correspond to? What about +?

What about an expression like (+ 3 (* 2 7))? The first part is the same, but when it comes to computing the value of the third expression, we find things are a little more complex: the third expression is not just a number, it’s a procedure-application-expression. So let’s work out its value.

Before we do, though, let’s again identify the entities: what sort of entities does the “3” correspond to? Answer: expression, number. What about (* 2 7)? Answer: expression, and procedure-application-expression, but not top-level-expression.

The + and 3 expressions will be evaluated exactly as in the previous example. When we reach (* 2 7), we see it is another procedure application expression, and once again apply the corresponding rules.

We proceed to evaluate the following expressions, in order. * is a name, so we must look it up in the top-level environment, where we find that it is bound to the built-in multiplication procedure. Moving on to the next expressions, we see that both 2 and 7 are numbers, which evaluate to themselves. We can now apply the built-in multiplication procedure to these arguments, and get the product of 2 and 7, which is 14.
So now (for the main expression) we have the addition procedure and the two values 3 and 14, with the 14 coming from the nested multiplication expression; we apply addition to these to get the sum 17, and write:

```
(+ 3 (* 2 7))
=> 17
```

I want to emphasize something important here: in a procedure application expression like

```
(+ 3 14)
```

all three items are expressions. In our case, the first happens to be a name and the second two are numbers, but the second two could be procedure-application-expressions as well. In general, it’s really easy to believe that the arguments to a procedure have to be simple items like 4 or "my string", but they’re far richer than that, and at least one homework problem you’ll soon encounter depends on this idea. (Of course, by that time we’ll have more kinds of expressions that can be dropped into these locations, which will make this idea more useful.)

**Question:** Is \((\text{define} \ \text{kilometers-per-mile} \ 1.6)\) a procedure-application-expression?

**Answer:** No. Per our rules, the first thing within the parentheses of a procedure-application-expression must be an expression. An expression can be a name, but a name cannot be a keyword.

**Question:** Is \((+ 17)\) valid? How about \((+ 17 18 19)\)?

**Answer:** Both are valid! This was an implementation decision made by the developers of Racket. They’ve noted things like this in the Racket documentation, which you can find by clicking here.

**Question:** Is \((17 18 19)\) valid?

**Answer:** Let’s be clear: that thing is a procedure-application-expression, because it consists of a sequence of expressions between parens, and that’s what makes for a proc-app-exp. So it is syntactically correct, as is the statement “Rocks cry urgently” in English. But when we apply the rules of evaluation to it, the first step is to evaluate the first expression (17); the resulting value is 17, which is not a procedure. The rules say that if the value is not a procedure, that’s an error and evaluation halts. So while it is syntactically correct, it is semantically invalid – it doesn’t mean anything, just like the English sentence “rocks cry urgently”, unless you’re fairly imaginative.

## 3 User-Defined Procedures

You can also define your own procedures, by writing something like this:

```
(define (f n) (+ n 1))
```
or more generally:

\[
(\text{define } (<\text{proc-name}> <\text{arg1}> ... <\text{argn}> ) <\text{body}> )
\]

where proc-name and all the args are names, and body is an expression.

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

The BNF for Racket is now updated to include <proc-defn>

\[
\begin{align*}
\text{<program>} &::= \text{<defn>} \ast [\text{<top-level-expr>}] \\
\text{<defn>} &::= \text{<name-defn>} \mid \text{<proc-defn>} \\
\text{<name-defn>} &::= (\text{define } <\text{name}> <\text{expression}> ) \\
\text{<proc-defn>} &::= (\text{define } (<\text{name}> <\text{name}*) <\text{expression}> ) \\
\text{<top-level-expr>} &::= \text{<expression>} \\
\text{<expression>} &::= \text{<number>} \mid \text{<string>} \mid \text{<name>} \mid \text{<bool>} \mid \text{<proc-app-expr>} \\
\text{<proc-app-expr>} &::= (<\text{expression}> <\text{expression}>* )
\end{align*}
\]

4 Summary

Ideas

- We now understand the difference between syntax and semantics. Like a sentence that can be syntactically correct, yet not make sense, a program can be syntactically correct, but have errors during processing. For example, (17 18 19) is three expressions within parentheses, a syntactically correct procedure-application-expression, but as 17 is not a procedure, this will cause an error during evaluation.

- We have started to build on the idea of environments. When definitions bind names to values, these bindings are placed in the current environment. For now, its enough to know that the current environment will start as the top-level environment and remain as the top-level environment until we add a few more rules in a week or two.

Skills

- Procedure-application-expressions have the form \((\text{proc \ arg1 ... argn})\) where each of proc, arg1, ..., argn is a Racket expression. They are evaluated as follows: (1) evaluate proc; the value must be a procedure or it’s an error; (2) evaluate all remaining expressions, known as arguments, to get their values; (3) apply the procedure to all values to get a resulting value, which is the value of the procedure-application-expression.

- We are starting to see how we can write useful programs using definitions and built-ins.
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)