Lecture 05: Functions and Procedures, Rules of Processing

10:00 AM, Sep 13, 2019

Contents

1 Announcements 1

2 The Racket Language 1

3 Processing 1

4 Evaluating 3

5 Summary 5

1 Announcements

- Go visit lonely Spike at office hours in CIT 473!

2 The Racket Language

We are going to start by introducing a few helpful functions that produce boolean results. They are called Predicates and can be distinguished because they often end with "?". There are around 300 to 400 built in predicates but you will only be using around 7 of them. Including predicates in the language arose from doing a process over and over again until they realized it was easier to just give that process a name. Some important predicates are:

\[
\text{(zero? 12)} = \text{false} \quad \text{(zero? 0)} = \text{true} \\
\text{(string? "hello")} = \text{true} \quad \text{(string=? "a" "b")} = \text{false}
\]

Briefly, onto sets: Two sets are the same if they have the same elements. Two sets are different if there is an element of one that’s not in the other.

In Racket we have headers that include input and output contracts. These are important because they tell the user what this program was built to handle. Defining the input restricts the domain, it makes a contract with the user of the program. The program doesn’t have to account for anything outside the specified domain, you aren’t promising the right answer for that. Even if your program still works on inputs outside the contract, don’t break your own contract by making check-expects with invalid input or using the function elsewhere with input not in the domain. The output can also be restricted and we call this the codomain.
3 Processing

Before we begin evaluating a program we have to understand it. I can’t really tell you what actually happens, I didn’t write DrRacket. But I can tell you a story of what is happening. I am going to give you a mechanism to imitate what Racket might be doing. It is going to be an abstract machine. It is a machine because it’s mechanistic- a fixed set of rules that describes how programs are processed.

Let’s recall our working BNF for CS17 Racket:

```
<program>  ::=  <defn>* [<top-level-expr>]
<defn> ::= <name-defn> | <proc-def>
<name-defn> ::= (define <name> <expression>)
<proc-def> ::= (define (<proc-name> <arg>*) <body>)
<proc-name> ::= <name>
<arg> ::= <name>
<body> ::= <expr>
<top-level-expr> ::= <expression>
<name> ::= <CS17name> | <othername>
<CS17name> ::= sequence of letters, digits, hyphens, starting with a letter, usually lowercase
<othername> ::= token consisting of non-special characters that can't be interpreted as a number and that isn't a keyword.
<expr> ::= <name> | <number> | <boolean> | <string> | <proc-app-expr>
<number> ::= anything that looks like a number
<boolean> ::= true | false
<string> ::= "any characters except double-quotes"
<proc-app-expr> ::= (<name> <expr>*)
```

Now that we are clear about our BNF, we can talk about the rules of processing. I read the program piece by piece, processing one piece at a time, when I encounter a name-definition, like `(define <name> <expr>)`, I Evaluate the expression to produce a value. Next, I Examine the top-level-environment (TLE) to see whether the name has already been defined; if so, it’s an error If not, I associate the name to the value v in the top-level environment. In class Spike drew out the TLE as a piece of paper. We say the name is “bound to” the value. When I examine the TLE (or any environment), I read from bottom to top.

Assuming the TLE starts out empty, what’s it look like after processing of the following? `(define a 3) (define b true)`

The word value is something I use to represent meaningful stuff. I might choose to represent the number we call “three” by putting three dots on my piece of paper. I might choose to represent it as a binary number “11” I might represent it as a piece of text: “three”.

An environment is a set of name value pairs. We have programs that read the text of a program and break it into small pieces then evaluate stuff that produces values and then prints out the actual text. This sequence is the REPL.

When I encounter a procedure definition of the form `(define (¡proc-name¿ ¡arg*¿ ) ¡body¿)` I enter the “proc-name” in the left-hand column of the TLE, and on the corresponding point in the right-hand side, I create a “closure”, my representation of a proc The closure is a rectangular box
containing the list of argument names and the body.

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!

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.”

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 $v$.

We can now write a slightly interesting program:

```
(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.

4 Evaluating

The rules of evaluation are still evolving. They tell us how to go from pieces of program text that are expressions to values. Let’s recall the BNF for expressions:
There are around seven, we will see some today.

1. a number-expression representing the ordinary number \( x \) evaluates to a number-value representing \( x \).

2. a string-expression representing the string \( s \) of characters evaluates to a string-value representing \( s \).

3. a bool-expression representing the boolean \( b \) evaluates to a bool-value representing \( b \).

4. a name, I look for the name in the environment (from bottom upwards) If it’s there and bound to some value, then is the value of the expression If it’s not there, it’s an error.

An expression that is an identifier \( \text{ident} \) is evaluated by checking whether the identifier is bound in the current environment; if not, it’s an error and evaluation and processing halt.

If the identifier 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.

The last rule of evaluation for now is that for procedure-application expressions. I’m now going to describe procedure-application expressions, which look like

\[
( <\text{expr}> <\text{expr}> ... <\text{expr}> )
\]

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

\[
( \text{proc} \ \text{arg1} \ \text{arg2} \ ... \ \text{argn} )
\]

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

1. Evaluate \( \text{proc} \); 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 \( \text{arg1} \ ... \text{argn} \) 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 a name, we follow rule 3 of our rules of evaluation. That is, 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 resulting 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.

## 5 Summary

### Ideas

- We now understand the Racket rules of Processing better.
- 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, it’s 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 \text{proc}, \text{arg1}, \ldots, \text{argn} is a Racket expression. They are evaluated as follows: (1) evaluate \text{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 CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)