(Provisional) Lecture 22: Rackette Overview, Binary Tree Analysis
10:00 AM, Oct 27, 2017

Contents

1 Announcements .................................................. 1

2 An OCaml Debugging Tip ...................................... 1

3 Introduction to Rackette ...................................... 1

3.1 Parsing .......................................................... 2

3.2 Processing/Evaluation ....................................... 3

3.3 Environments, Definitions, and Identifiers ............... 3

4 Balanced Binary Trees and Big-O ................................ 4

5 Summary ............................................................ 5

Objectives

By the end of this lecture, you will:

- have a broad overview of the Rackette project
- have analyzed the contains7? procedure

1 Announcements

This upcoming Friday you’ll submit Eliza, after which you’ll be assigned a new project, Rackette. In this lecture, we’ll introduce many of the concepts you’ll need to tackle to successfully complete Rackette.

2 An OCaml Debugging Tip

Unlike Racket, OCaml doesn’t come with a built-in interaction window. Instead, to see if our code is working, we compile our OCaml program through the terminal. As you’ve probably seen, it will spit out some text, as well as error messages if applicable. However, OCaml’s error messages often seem jumbled and difficult to read, especially when we’re trying to implement many different procedures in a single .ml file.
To isolate errors, we can instead run the `ocaml` command in the terminal, allowing us to use the interactive version of OCaml, directly in the terminal.

We can now copy and paste one chunk of our code in at a time (likely one procedure at a time), running each in the terminal separately, and seeing which errors it generates on its own. If something goes wrong, the OCaml program will even underline for you the place where it first got confused.

This is a useful debugging technique because we can better pinpoint what’s going wrong in our code, or at least get a clearer sense of where to start looking.

### 3 Introduction to Rackette

Most of today’s lecture notes will be on Rackette. There are two reasons we need to start this. You’re going to see how some of what you’ll doing in Eliza will be relevant for the next project. Also, you have a lab this weekend that does a little part of Rackette as a warmup. We want to give you an introduction in class first.

Let’s start with the main `rackette` program. The type signature for the main `rackette` procedure will be: `rackette : string -> string`.

It will take in Racket code in the form of a string, and output what the interactions pane of DrRacket would output, likewise in the form of a string.

A typical input to Rackette may be:

```racket
(define (f x) (+ x 1))
(f 3)
```

Note, however, you are writing a Rackette interpreter, not a Racket interpreter; there are some simplifications in Rackette. For example, the only numbers are integers. You don’t have to include `let*` (but it’s fun!). You also don’t have to include every “built-in” that we’ve used in Racket. Built-ins such as `+` need to only take two arguments in your implementation. You will implement `if`-expressions, `cond`-expressions, `lambda`-expressions, and procedure-application expressions. There will probably be more simplifications that you’ll see later.

The first step in breaking down something that takes in text as input is a process calling parsing.

### 3.1 Parsing

Parsings involves taking a sentence or a program and identifying the major parts.

In the case of a sentence, you might identify the subject and predicate, for example. In Rackette, “parsing” means looking at the input text and seeing whether it matches the BNF for Rackette. There are really cool tools that take BNF as input, and produce a parser as output. But, we’re not going to use one of those — you’re going to write a Rackette-specific parser yourself!

Experience shows that this is best done as a multi-step process. First, check that parens match, and identify numbers, strings, and non-number entities. Then, parse the resulting things to be sure they’re legal Rackette. When we describe this, we’re essentially talking about a language whose BNF looks like this:
Parsing ensures that the provided string looks somewhat like our BNF. To simplify this, we’ve written a procedure for you called read. Its type signature is `read : string -> racket_text`. [We might change one of the type names, like `racket_text`, before the Rackette project is handed out.]

Here’s a sample input and output for `read`:

Input: "(define (f x ) (+ x 1)) (f 3)"
Output: `[List [Ident "define" ; List [ Ident "f" ; Ident "x"] ; List [Ident "+" ; Ident "x" ; Num 1]] ; List[Ident "f" ; Num 3]]`

Your task will be to take something like this and essentially ask, “Is that a legal Rackette program?” Each item in a Rackette program is either a definition or an expression. So specifically, we’ll want to ask: does that first thing in the list match the notion of a definition? We know that a definition should begin with the keyword `define`, followed by an identifier or a list. Our example output does in fact begin with `define`, and is followed by a list! (And, as you probably guessed, you can use pattern matching to check this in your implementation!)

### 3.2 Processing/Evaluation

Once we’ve figured out that we have a definition followed by a procedure application expression, we need to process/evaluate these. Take the above example:

```
(define (f x) (+ x 1))\n(f 3)
```

Processing the first line will add a new “binding” to the top-level environment. Evaluating the second line will require looking up bindings for “f” and “+” in the top level environment, and doing some arithmetic. The good news is that OCaml knows how to do arithmetic!

### 3.3 Environments, Definitions, and Identifiers

An environment is a list of all the associations of identifiers to values produced by a definition, or in other words, a list of bindings.

We’ll be doing “lookups” and “extending” environments with new bindings. We also need to create an empty environment, and determine how to add a binding to an environment.

In order to do these things, we first have to figure out how to represent a value. A value could be an integer, string, boolean, closure, or a builtin. That suggests a type definition with multiple options. For now, we’ll just say there are two possible values, namely `Int of int | String of string`, because that captures all the subtleties we’ll need.
Now, let’s discuss *identifiers*, or as we’ve been calling them, names. First, let’s think of an example of an identifier — anything from `list1` to `+` works. For instance, the identifier `+` is bound to a built-in procedure — specifically, the built-in procedure for addition. Regardless, we come to the conclusion that identifiers are simply strings.

Finally, we need to figure out how to represent a binding. A *binding* relates an identifier to its value. So, we could represent a binding as a tuple of an identifier and value.

To summarize, here are the the types we’ve identified so far.

```plaintext
type identifier = ID of string
type binding = identifier * value
type value = Int of int | String of string
type environment = binding list;
```

## 4 Balanced Binary Trees and Big-O

### Analysis for `contains7?` for a balanced tree

Recall the `contains7?` procedure we wrote that searched for 7 within a binary tree. For the purposes of this analysis, let’s look at `contains7?` for a balanced tree.

Let \( C(d) \) be the largest number of elementary operations in the evaluation of `contains7?` on any input tree of depth \( d \).

Normally, we say on “any input of size \( n \).” Here, I’m saying “on any input tree of depth \( d \).” This’ll simplify the process, and you’ll soon see why.

As always, we’ll start with our base case. How long does it take to see if a tree of depth 0 contains the number 7? Essentially, it is a constant time operation. The only possibility is a leaf which doesn’t contain 7, or a node which may or may not contain 7. Let us say:

\[
C(0) = A
\] (1)

In our general recursive case, we have to see if the node is 7, and if it isn’t, we have to look in one of the subtrees. We do a little bit of constant work checking if the node we are at is 7, and we recur on one of the subtrees. Thus, we can say:

\[
C(d) \leq B + C(d - 1)
\] (2)

This looks pretty similar to the recurrence relations we’ve previously proved. We can say \( C \in O(n \rightarrow n) \).

Now, we need to determine what the depth of a tree with \( n \) nodes actually is. It could be hugely unbalanced—— the left sub-tree of all nodes could be a leaf, and the right sub-tree could be a node for all but the last node.

We’ll show that if you have a balanced tree with no more than \( n \) nodes, the tree has depth less than \( 1 + \log(n) \). This is equivalent to showing that a balanced tree of depth \( d \) has at least \( 2^{d-1} \) nodes, as this statement is the contrapositive of the previous. Let’s prove it.

Let \( S(d) \) be the smallest number of nodes in any balanced tree of depth \( d \). We claim \( S(d) = 2^{d-1} - 1 \).
$S(1) = 0$, as a tree of depth 1 contains only a single leaf, or in other words, no nodes. Hence our claim is true for $d = 1$.

Suppose a tree of depth $k$ contains fewer than $2^{k-1} - 1$ nodes, and $k$ is the smallest number for which this occurs. Then $k \neq 0$ and $k \neq 1$, because there are no depth-0 trees, and depth-1 trees all contain 0 nodes.

Each child contains at least $2^{k-2} - 1$ nodes. Then, we must account for the top node by adding one.

We now have a total of $1 + 2(2^{k-2} - 1) = 1 + 2^{k-1} - 2 = 2^{k-1} - 1$ nodes. The tree contains at least $2^{k-1} - 1$ nodes. This is a contradiction. As such, our claim must hold for all number of nodes.

Depth $d$ implies about $2^d$ nodes in a balanced tree. If we set $n = 2^d$, then the depth can be represented by about $\log(n)$. So, having $n$ nodes corresponds to having a depth of about $\log(n)$ (plus or minus one, and hence searching in a binary-search-tree has a runtime in $O(n \mapsto \log n)$, which is awesome!

## 5 Summary

### Ideas

- A binding associates an identifier with a value.
- An environment is a collection of associations of identifiers to values (bindings) produced by a definition. We can represent environments as lists of bindings.
- We’ve begun looking into the process of parsing, processing, and evaluating.

### Skills

- We have learned a helpful way to debug our OCaml procedures.
- We have gone through a brief overview of key concepts in the Rackette project.
- We have analyzed contains?, a procedure that recurs on balanced binary trees.

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/cs17/feedback](http://cs.brown.edu/courses/cs17/feedback)