(Provisional) Lecture 09: Recursive Expression Evaluation
10:00 AM, Sep 23, 2019

Contents

1 Warning 1

2 A Few Things to Note 1

3 Step-by-Step Evaluation (of a Complex Expression) 1

4 Anyone got a 17? 2

5 What About a Recursion Example? 4

6 More Recursion 8

7 How Fast or Slow is my Program? 9

8 Operation Counting 9

9 Summary 10

1 Warning

These provisional notes are mostly correct, and mostly follow the language used in class, but there are some differences, and the lecture TA will be modifying them over the next day or two. If something seems screwy, look at the revised notes to see if that clears it up. And in general, I hope that the organization of the day’s Powerpoint slides will work well for you doing your own recursive evaluations. –Spike

2 A Few Things to Note

At any time in the course, you’ll have encountered certain aspects of Racket. Your homework is to use only those aspects of racket, not others. For instance, if the homework had asked you to produce a procedure that finds the length of a list, you should not write

```
(define mylen length)
```

because at this point, you have not encountered the built-in length.
Note: For these lecture notes, please refer to the lecture slides posted online to see the evaluation step-by-step.

For the tables in these lecture notes:

- The first column contains an expression we want to evaluate.
- The second column (labeled Exp type.) will tell us what type of expression we’re evaluating.
- The third column (labeled Env.) contains the relevant parts of our current environment.
- A red color in the Value (fourth) column means we haven’t figured out the value of our expression yet (green will mean we have evaluated the expression and know the value.)

3 Step-by-Step Evaluation (of a Complex Expression)

To run the Rules of Evaluation, we have to look at an expression and decide what kind of expression it is (this is where the second column comes in handy!).

Steps in evaluating a “complex” expression:

1. What kind of expression are we trying to evaluate?

2. Evaluate relevant subparts of the expression. For proc-app-expressions, that means both the first, which should evaluate to a procedure, and all the others. For things like if-expressions and or-expressions, short circuiting may involve only evaluating a few subparts.

3. Once you have values for the relevant “inner” expressions, you can evaluate the “complex” expression which is no longer complex!

Note: Look at the lecture slides for why list-length works.

4 Anyone got a 17?

We wrote contains17? last class, which takes an int list tests whether a list of integers contains the number 17, and produces a boolean.

You should see a pattern, namely, that the input list contains a 17 in one of two situations:

- if the rest of the list contains a 17, or
- if not, but if the first item in the list happens to be a 17.

Thus, if the Recursive Input is Original Output will be true.
If the Recursive Output is false, but (first Original Input) is 17, Original Output is true. Otherwise Original Output is false.

From this, and the design recipe, you should produce code that looks like this
;; Data Description:
;; An int list is either
;; empty
;; (cons n lst) where n is an int and lst is an int list
;; nothing else is an int list
;; Examples:
;; int: 0, 3, -2
(define lst0 empty)
(define lst1 (cons 17 empty))
(define lst2 (cons 17 (cons 4 empty)))
(define lst3 (cons 3 (cons 17 empty)))
(define lst4 (cons 3 (cons 1 empty)))
;;
;; contains17? : (int list) -> bool
;;
;; input: aloi, a list of integers
;; output: true if aloi contains 17; false otherwise
;;
;; Recursion Diagram
;; Original Input:(cons 17 empty)
;; Recursive Input: empty
;; Recursive output: true
;; Original Output: true
;;
;; Original Input:(cons 17 (cons 4 empty))
;; Recursive Input: (cons 4 empty)
;; Recursive output: false
;; Original Output: true
;;
;; Original Input:(cons 3 (cons 17 empty))
;; Recursive Input: (cons 17 empty)
;; Recursive output: true
;; Original Output: true
;;
;; Original Input:(cons 3 (cons 1 empty))
;; Recursive Input: (cons 1 empty)
;; Recursive output: false
;; Original Output: false
;;
(define (contains17? aloi)
  (cond
   [(empty? aloi) false]
   [(cons? aloi) (if (contains17? (rest aloi))
                   true
                   (if (= 17 (first aloi))
                       true
                       false))]]))

(check-expect (contains17? lst0) false)
(check-expect (contains17? lst1) true)
(check-expect (contains17? lst2) true)
(check-expect (contains17? lst3) true)
(check-expect (contains17? lst4) false)

This is a correct, but ugly program. A Racket programmer would look at it and wonder why it
looked like that. How come? We have the first if expression returning a bool, and the second if expression returning two bools.

Let’s look at that last bit:

```
(if (= 17 (first aloi))
  true
  false)
```

Suppose that the first item in aloi is 17. What’s the value of the if-expression? It’s true, right? Now ask yourself: what’s the value of the “condition” part of the if-expression, i.e., of (= 17 (first aloi))? It’s also true.

Now suppose that the first item is not 17. Then the whole if-expression evaluates to false, but so does just the condition expression.

So we can replace the whole if-expression by just the condition! Our second cond case now looks like this:

```
(define (contains17? aloi)
  (cond
   [(empty? aloi) false]
   [(cons? aloi) (if (contains17? (rest aloi))
                   true
                   (= 17 (first aloi)))]))
```

We’re not done yet!

We’ve now got a situation in which we have two conditions, and if either one of them is true, the value we want is true; otherwise we want false. Well, that’s exactly what or provides. We can rewrite:

```
(define (contains17? aloi)
  (cond
   [(empty? aloi) false]
   [(cons? aloi) (or (contains17? (rest aloi))
                    (= 17 (first aloi)))]))
```

Finally, suppose we have a list of 1000 items, and the first one is 17. Do we need to look at the other 999? Heck, no! So because of the way that or short-circuits, we should swap the order

```
(define (contains17? aloi)
  (cond
   [(empty? aloi) false]
   [(cons? aloi) (or (= 17 (first aloi))
                   (contains17? (rest aloi)))]))
```

Now that is idiomatic Racket code!
5 What About a Recursion Example?

Remember that when we defined list-length (which we’re now referring to as len for short) the result of that definition is that the identifier len is bound to a closure—a closure in which the argument list is lst and in which the body is a cond expression. For the remainder of our notes (and in the slides), this closure will be called C1.

```scheme
;; example ints
;; 1
;; 0

;; len: (int list) -> int
;; Input: a list of integers, aloi
;; Output: an integer, the length of aloi
(define len aloi)
  (cond
   [(empty? aloi) 0]
   [(cons? aloi) (+ 1 (len (rest aloi)))]
)

;; An example of using the len procedure
(len (cons 1 empty))

;; ^ - We'll refer to this expression as A
;; [_____________] - We'll refer to this expression as B
```

Evaluating (len (cons 1 empty)):

1. First, we evaluate A, namely len to see that the result is a closure, C1, a kind of procedure value, so overall we’re working with a proc-app-expression.

2. Evaluate B, also a procedure application, which evaluates to the list: (cons 1 empty).

3. Evaluate the body of C1 in an environment consisting of the TLE, extended by new bindings in which the formal arguments are bound to the actual arguments:
   (a) Evaluate the cond expression by looking at each condition one by one (in order).
   (b) Evaluate each condition and when you hit one that evaluates to true:
       (c) ...evaluate the corresponding result expression in the same context (environment).
       (d) Repeat (1-3) (since (3) is recursive) until you get a final value.

Visually, this process of evaluation for (len (cons 1 empty)) will look something like this:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Exp Type</th>
<th>Envt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>user-defined proc</td>
<td>Top Level Environment</td>
<td>C1</td>
</tr>
</tbody>
</table>
2. Evaluate B, also a procedure application, which evaluates to the list: \((\text{cons} \ 1 \ \text{empty})\).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Exp Type</th>
<th>Envt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{len})</td>
<td>user-defined procedure</td>
<td>Top Level Environment</td>
<td>C1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>identifier</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>len</td>
<td>C1</td>
</tr>
<tr>
<td>((\text{cons} \ 1 \ \text{empty}))</td>
<td>procedure application</td>
<td>...</td>
<td>((\text{cons} \ 1 \ \text{empty}))</td>
</tr>
</tbody>
</table>

3. Evaluate the body of \(C1\) in the Top Level Environment, extended by the binding, where the formal arguments are bound to the actual arguments. Recall that our closure, \(C1\), represents a procedure. Visually, it looks something like:

```
args:   aloi
body:

(cond
  [(empty? .....)]
  [(cons? .....)])
```

Note that “args” corresponds to any inputs to the procedure, in this case \(\text{aloi}\), and “body” corresponds to the unevaluated expression which constitutes the body of our length procedure. In this case, the body is a \(\text{cond}\) expression which produces one result if the list is a \(\text{cons}\), and another if it’s \(\text{empty}\).

Now it’s time to evaluate!

To do so, we extend our Top Level Environment by adding a new, Local Environment, where the formal arguments have been bound to the actual arguments. This local environment is only temporary, and will only exist for as long as it takes for the body to be fully evaluated. So, we now have:

```
Top Level Environment

<table>
<thead>
<tr>
<th>identifier</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>C1</td>
</tr>
</tbody>
</table>
```

and,

```
Local Environment

<table>
<thead>
<tr>
<th>identifier</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>aloi</td>
<td>(cons 1 empty)</td>
</tr>
</tbody>
</table>
```

In this case we have \(\text{aloi}\) as our formal argument and \((\text{cons} \ 1 \ \text{empty})\) as our actual argument. The actual arguments are bound to the formal arguments temporarily”. So, as we now start to evaluate the body of the closure, we will look up the values of any identifiers we find in these two environments. Racket will first look in the Local Environment for the binding, and, if the identifier was not found, continue searching in the Top Level Environment.

(a) Evaluate the \(\text{cond}\) expression by looking at each condition one by one (in order).

(b) Evaluate each condition and when you hit one that evaluates to \(\text{true}\):

(c) ...evaluate the corresponding result expression in the same context (environment).

Following these next three steps, we look up \(\text{aloi}\) in the environments (in the order outlined above), and find that \(\text{aloi}\) is indeed bound to \((\text{cons} \ 1 \ \text{empty})\) in the Local Environment.
So, finding that we have a **cons** list, we go on to evaluate the corresponding result expression in the same context.

Visually, this looks something like:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Exp Type</th>
<th>Envt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 1 (len (rest aloi)))</td>
<td>procedure application expression</td>
<td>See environments above</td>
<td>?</td>
</tr>
</tbody>
</table>

Following the rules of evaluation for evaluating a procedure application expression, we evaluate it one expression at a time.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Exp Type</th>
<th>Envt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 1 (len (rest aloi)))</td>
<td>procedure application expression</td>
<td>See environments above</td>
<td>?</td>
</tr>
<tr>
<td>+</td>
<td>builtin procedure</td>
<td></td>
<td>Closure</td>
</tr>
<tr>
<td>1</td>
<td>number</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>(len (rest aloi))</td>
<td>procedure application</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

In the last step, Racket recognizes that `(len (rest aloi))` is in fact a procedure application expression. So following the rules of evaluation, going through one piece at a time:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Exp Type</th>
<th>Envt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>user-defined procedure</td>
<td>See environments above</td>
<td>C1</td>
</tr>
</tbody>
</table>

Just as above, looking up the identifier `len` in our Top Level Environment (extended by the local environment) gave us the closure `C1`, since that binding remains in the Top Level Environment.

Now all that’s left is to evaluate the actual arguments given to our user-defined procedure, `(rest aloi)`. Remembering that we have to look up the value of `aloi` in our Top level Environment extended by our Local Environment, we get:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Exp Type</th>
<th>Envt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(rest aloi)</td>
<td>...</td>
<td>Local Environment</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>identifier</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aloi</td>
<td>(cons 1 empty)</td>
</tr>
</tbody>
</table>

Since `(rest (cons 1 empty))`, (i.e. rest of what we get when we look up `aloi`), is `empty`, this will give us `empty`.

(d) Repeat (1-3) (since (3) is recursive) until you get a final value.

Now knowing that we are invoking the `len` procedure on an empty list, we follow the exact same steps as we do above when we invoked `len` on a cons list.

Namely, we evaluate the body of the closure `C1` in an environment consisting of the TLE plus a local environment, where the formal arguments are bound to the actual arguments.

We now have:

<table>
<thead>
<tr>
<th>Top Level Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
</tr>
<tr>
<td>len</td>
</tr>
</tbody>
</table>

and,

<table>
<thead>
<tr>
<th>Local Environment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
</tr>
<tr>
<td>aloi</td>
</tr>
</tbody>
</table>
When looking up an identifier in these environments, we start with the most recent, and work our way down. So, when looking up any identifiers in this case, we'd start with Local Environment 1, then the Top Level Environment.

You can think of local environments like index cards - each time you add a new one, you stack it on top of the old ones, and always look in the top-most index card first when looking up identifiers.

Now, again we

(a) Evaluate the **cond** expression by looking at each condition one by one (in order).
(b) Evaluate each condition and when you hit one that evaluates to **true**: 
(c) ...evaluate the corresponding result expression in the same context (environment).

In this case, when we go to look up aloi, we find that it’s **empty**! So when we evaluate the corresponding result expression for the appropriate **cond** case, we just get 0.

Note that, once our closure has returned a value and the procedure has terminated, the local environment which had the temporary bindings between the formal and actual arguments for that procedure goes away.

So, after 0 is returned, we are back to evaluating the first use of **len** and the environment looks like this:

<table>
<thead>
<tr>
<th>Top Level Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
</tr>
<tr>
<td>len</td>
</tr>
</tbody>
</table>

and,

<table>
<thead>
<tr>
<th>Local Environment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
</tr>
<tr>
<td>aloi</td>
</tr>
</tbody>
</table>

Knowing now what \((\text{len (rest aloi)})\) evaluates to, we can go back and update our table from before!

<table>
<thead>
<tr>
<th>Expression</th>
<th>Exp Type</th>
<th>Envt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 1</td>
<td>builtin procedure</td>
<td>&quot;&quot;</td>
<td>Closure</td>
</tr>
<tr>
<td>(len (rest aloi))</td>
<td>procedure application</td>
<td>&quot;&quot;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

And, now that we know the value of everything in our procedure-application expression, we can evaluate the procedure-application expression as a whole!

<table>
<thead>
<tr>
<th>Expression</th>
<th>Exp Type</th>
<th>Envt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 1 0)</td>
<td>procedure application</td>
<td>See environments above</td>
<td>1</td>
</tr>
</tbody>
</table>

Once again, now that our procedure is done and has returned a value, the local environment which contained the temporary bindings between its formal and actual arguments (in this case, Local Environment 1) will disappear, leaving us only with the Top Level Environment.

When in doubt, follow these rules of evaluation to find the result of any recursive procedure!
6 More Recursion

While evaluating these functions, you may be wondering why recursion even works. Specifically, why we can call a function without having fully defined it. The answer lies in how Racket programs are actually evaluated— the Rules of Evaluation! Consider the \texttt{len} procedure. \texttt{len} is bound to a closure with argument \texttt{aloi} and body which is the body of the procedure. The trick is that this body is not evaluated at the time of adding the binding, it is simply put in the Top Level Environment as it is. So every time we call \texttt{len}, it is going to find a binding to a closure, and evaluate it normally.

Note: Most of the examples from the slides are applicable to all types of data. The procedures written in class were for numbers, but here, the same procedures will be generic (if possible) so you can distinguish small (but very important!) stylistic and functional differences.

**SUPER Important Note:** All lists in CS17 are homogeneous, i.e., must contain elements of the same data type. For example, \texttt{(cons 3 \texttt{(cons 0 empty))}} is a perfectly acceptable list. As are \texttt{empty} and \texttt{(cons "CS"\texttt{(cons "17"\texttt{(cons "Rocks"empty))}}). However, \texttt{(cons "CS"\texttt{(cons 17 \texttt{(cons "Rocks empty))}})} is an unacceptable list, despite how true that statement is. Notice the difference between \texttt{(cons "CS"\texttt{(cons 17 \texttt{(cons "Rocks empty))}})} and \texttt{(cons "CS"\texttt{(cons 17 \texttt{(cons "Rocks empty))}})}— in the former, \texttt{"17"} is a \texttt{string} as are the other two elements, in the latter, \texttt{17} is an \texttt{integer} while the other two elements are \texttt{strings}.

7 How Fast or Slow is my Program?

When we have two functions, for example a linear and an exponential function, we’ve seen how to characterize them as eventually larger or smaller. We can do this when we have a mathematical relation describing the functions. When we want to use this technique to evaluate how fast or slow a computer program is, how would go about this? We would need to find a recurrence relation, which is some mathematical relation describing the program which we can solve to show how fast the program is, without really finding the exact function. We will be learning about how to do this in detail in the coming weeks!

Faster programs were much more important in the past when computers were expensive, but recently the focus has shifted somewhat to how easy a program is to maintain. However, as big programs have to deal with more and more data, speed becomes extremely important. Now the important question is— for bigger data sets, does our program eventually start getting faster? Because if we talk about any fixed sized data set, we know that eventually our program will get fast, with computers becoming increasingly powerful. But we will always keep getting more data, faster than we can keep up with. This is where programs that are fast in the long run become important.

Some more things to keep in mind as we start talking about this is that there are other factors, unrelated to how good your code is, that can affect how long your program takes to run. Because people code in different languages, use computers with different processors, and other factors, we ignore constants while evaluating the speed of programs. If one program takes twice as long as another— from our point of view, they are equally fast. There are two reasons for this— firstly, practically it doesn’t matter as it does not cause a large effect in the long run, and secondly, it makes the math much easier!
8 Operation Counting

Let’s count how many operations are needed to evaluate a couple expressions. The operations cons, first, rest, +, -, *, /, empty?, cons?, or, and, =, binding a name to a value, looking up a procedure, evaluate a num, bool, string, empty takes constant time to operate, i.e it takes 1 operation count.

1. Let’s look at (+ 3 5). How many operations does this take?
   (a) 1 for evaluating 3
   (b) 1 for evaluating 5
   (c) 1 for looking up + in the Top Level Environment.
   (d) 1 for actually performing the addition operation.

   Therefore, the operation count for (+ 3 5) is 4.

2. Let’s now look at (contains17? empty). How many operations does this take?
   (a) 1 for looking up contains17?
   (b) 1 for evaluating empty
   (c) 1 for binding aloi to empty
   (d) 1 for evaluating empty?
   (e) 1 for evaluating aloi
   (f) 1 for performing the operation empty? on aloi
   (g) 1 for evaluating false

   Therefore, the operation count for (contains17? empty) is 7.

Now that we have evaluated the base case for contains17?, we can approximately guess the operations for an element with a one element list to be 18. This includes the base case scenario. Thus we can generalize the operation counting for the list to be 11n + 7 where 7 was our base case operation count.

9 Summary

Ideas

- When we are evaluating a procedure application expression, we always start by extending our environment with a new, local environment, which binds the formal arguments of the procedure to the actual arguments that the procedure is being applied to. This local environment will disappear once our procedure-application has been successfully evaluated (i.e., we’ve followed the logic of the body of the procedure and determined the correct value to return).

- We know that, in CS 17, lists are homogeneous (i.e. can only contain items of the same data types).

- We know how to write a recursive procedure that will check to see if an input list contains the number 7.
Skills

- We’ve learned how to use tables to break down the evaluation of a recursive procedure. That is, once we know we are dealing with a procedure application expression, we know how to look up the appropriate identifiers in our environments in the correct order (i.e. looking in chronological order, with the most recent local environments being first, and the top level environment being last) and follow the rules of evaluation until we reach a base case. Then, we take that result, and retrace our steps through our recursive calls to produce one final result.

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).