(Provisional) Lecture 09: Recursive Expression Evaluation
10:00 AM, Sep 25, 2017

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 Summary 8

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

\[
\text{(define mylen length)}
\]

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!

4 Anyone got a 17?

Let’s write contains17?, which tests whether a list of integers contains the number 17.

You should draw a recursive diagram for each of four inputs:

(cons 17 empty)  
(cons 17 (cons 4 empty))  
(cons 3 (cons 17 empty))  
(cons 3 (cons 1 empty))

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.

From this, and the design recipe, you should produce code that looks like this

```scheme
;; 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
;;
(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)

Now we have to fill in the middle bit, and from the logic above, we can write this:

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

This is a correct, but ugly program. A Racket programmer would look at it and wonder why it looked like that. How come? Well, 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
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.

```
;; 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 3 empty))
;; ^ - We'll refer to this expression as A
;; [_____________] - We'll refer to this expression as B
```
Evaluating \((\text{len } (\text{cons } 3 \text{ empty}))\):

1. First, we evaluate A, namely \text{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: \((\text{cons } 3 \text{ empty})\).

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

Visually, this process of evaluation for \((\text{len } (\text{cons } 3 \text{ empty}))\) will look something like this:

1. Evaluate A, which is a user-defined procedure application. The first step is to evaluate \text{len}, which is bound to a closure, C1.

<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 value</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{len} C1</td>
<td></td>
</tr>
</tbody>
</table>

2. Evaluate B, also a procedure application, which evaluates to the list: \((\text{cons } 3 \text{ empty})\).

<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 value</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{len} C1</td>
<td></td>
</tr>
<tr>
<td>\text{(cons } 3 \text{ empty)}</td>
<td>procedure application</td>
<td>...</td>
<td>\text{(cons } 3 \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:

   \begin{verbatim}
   args: aloi
   body:
   \begin{verbatim}
   (cond
   [(empty? .....)]
   [( cons? ..... )] )
   \end{verbatim}
   \end{verbatim}

   Note that “args” corresponds to any inputs to the procedure, in this case 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:

<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</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
</tr>
<tr>
<td>aloi</td>
</tr>
</tbody>
</table>

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 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).

Following these next three steps, we look up aloi in the environments (in the order outlined above), and find that aloi is indeed bound to (cons 3 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 3 empty)</td>
</tr>
</tbody>
</table>

Since (rest (cons 3 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:
Knowing now what (\text{len} (\text{rest} \text{ 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>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!

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

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, (\text{cons} 3 (\text{cons} 0 \text{ empty})) is a perfectly acceptable list. As are \text{empty} and (\text{cons} "\text{CS}"(\text{cons} "\text{17}"(\text{cons} "\text{Rocks}\text{empty}"))). However, (\text{cons} "\text{CS}"(\text{cons} 17 (\text{cons} "\text{Rocks}\text{empty}"))) is an unacceptable list, despite how true that statement is. Notice the difference between (\text{cons} "\text{CS}"(\text{cons} "\text{17}"(\text{cons} "\text{Rocks}\text{empty}"))) and (\text{cons} "\text{CS}"(\text{cons} 17 (\text{cons} "\text{Rocks}\text{empty}")))—in the former, "17" is a \text{string} as are the other two elements, in the latter, 17 is an \text{integer} while the other two elements are \text{strings}.

7 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 CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)