(Provisional) Lecture 13: More Analysis and Intro to Lambda
10:00 AM, Oct 2, 2019

Contents

1 Announcements ............................. 1
  1.1 Piazza .................................. 1
  1.2 Course Missive ......................... 1
  1.3 Type signatures ......................... 1
  1.4 Design recipe .......................... 1
  1.5 Non-overlapping cond cases .......... 1

2 Numbers .................................... 2
  2.1 Approach 1 (using length) .......... 2
  2.2 Approach 2 (incrementing) ........ 3
  2.3 Approach 3 (using a helper) ....... 3

3 Censor ...................................... 4

4 Higher Order Procedures .................. 5

5 Lambda ..................................... 6

6 Summary ................................... 7

1 Announcements

1.1 Piazza

Piazza, which you saw in lab 1, is an essential tool for this course. Check it often. Updates to assignments to clarify stuff appear there. Don’t miss them!

1.2 Course Missive

You’re supposed to have read the course missive. There’ll be a quiz about it on Wednesday, one question, something basic, and not too detailed (i.e., not “at what exact time do Spike’s office hours on Tuesday end?”).
1.3 Type signatures

I sometimes, writing about things, say things like “aloi is an int-list,” because writing “aloi is an (int list)” makes it look as if the type I’m talking about is a parenthetical expression in the sentence, despite the typography. But in type signatures, you should always use (int list), in parenthesis, with no hyphen, to indicate such a thing. We’ll start deducting points if you fail to do so.

In the same way, “arrow types”, like (int -> bool), should be parenthesized.

1.4 Design recipe

You should follow the design recipe for every procedure. And from now on, you get no credit for doing it right, just as you get no points for correct spelling on a history essay: hand in a perfectly-spelled review of the My Little Pony movie instead of an analysis of William Jennings Bryant’s Cross of Gold speech, and you’ll get a zero, right? Same thing goes for the design recipe from now on: we expect you to follow it, and we’ll deduct points if you don’t.

1.5 Non-overlapping cond cases

Your cond cases should not overlap; if they do, and you swap the order of the cond cases in your program, it’ll function differently. See today’s slides for an example.

There may be some rare case where checking for exclusivity is computationally very expensive; in that case, you can add a comment explaining why the order matters. I don’t believe that this will happen, but I want to allow reasonable choices if it does for some reason.

2 Numbers

We’re going to look at a silly problem — take a list of n items, and convert it to a decreasing sequence of the n integers from n down to 1. We will call this procedure numbers. Here are some examples:

(numbers (list "a" "b" "c")) => (list 3 2 1)  
(numbers empty) => empty  
(numbers (list 4 8 -2 3 1 5)) => (list 6 5 4 3 2 1)

As per usual, to get going on this procedure we’ll begin with a recursive diagram.

Orig. input: (list "a" "b" "c")  
    Rec. in: (list "b" "c")  
    Rec. out: (list 2 1)  
...  
Orig. output: (list 3 2 1)

We’re going to look at three different approaches to this problem:
1. Cons the length of the input onto the front of the recursive output
2. Take the first of the recursive output, increment it, and cons that onto the recursive output
3. Do the same thing, but with a helper procedure.

Let’s try each one.

2.1 Approach 1 (using length)

This approach cons-es the length of the input list onto the front of the recursive output:

\[
\text{define (numbers alod)} \begin{align*}
\text{cond} \{ & \text{[empty? alod) empty]} \\text{[cons? alod) (cons (length alod) (numbers (rest alod)))]} \}
\end{align*}
\]

Now, let’s find the recurrence relation for numbers:

\[
\begin{align*}
C(0) &= A \\
C(n) &= B + (P + Qn) + C(n - 1), \text{ for } n > 0.
\end{align*}
\]

Finally, through plug-n-chug we get:

\[
C(n) = Sn^2 + Tn + U
\]

From this, we gather that this approach runs in \(n^2\) time, which isn’t great.

2.2 Approach 2 (incrementing)

This approach takes the first of the recursive output, increments it, and cons-es it onto the recursive output.

\[
\text{define (numbers alod)} \begin{align*}
\text{cond} \{ & \text{[empty? alod) empty]} \\text{[empty? (rest alod)) (list1)]} \\text{[cons? alod) (cons (+ 1 (first (numbers (rest alod)))) (numbers (rest alod)))]} \}
\end{align*}
\]

Now, let’s find the recurrence relation for numbers:

\[
\begin{align*}
C(0) &= A \\
C(1) &= B \\
C(n) &= D + 2C(n - 1), \text{ for } n > 1.
\end{align*}
\]

Finally, through plug-n-chug we get something that runs in \(2^n\) time, which is way worse than Approach 1!

What can we conclude about our attempts thus far?
• Using \texttt{length} inside a recursion is bad
• Recurring twice is worse (though sometimes absolutely necessary – but not in this case!)

Below are two recurrence relations, one for each function. Which function has smaller runtime? How do you know?  
\begin{align*}
C(0) &= A \\
C(n) &= B + C(n - 1) \text{ for } n \geq 0, \ B > 0. \\
F(0) &= G \neq 0 \\
F(n) &= H + F(\text{floor}(n/2)) \text{ for } n \geq 0
\end{align*}

2.3 Approach 3 (using a helper)

This approach is the same as Approach 2, but it uses a helper procedure, \texttt{build}.

\begin{verbatim}
;; build: (int nelist) -> (int list)  ;; a helper for numbers; cons 1 plus the first item of aloi onto aloi
(define (build aloi)
  (cond
   [(cons? aloi) (cons (+ 1 (first aloi)) aloi)]))

;; numbers: ('a list) -> (int list)  ;; Input: a list, alod, of n items  ;; Output: a decreasing sequence of n integers from n down to 1.
(define (numbers alod)
  (cond
   [(empty? alod) empty]
   [(empty? (rest alod)) (list 1)]
   [(and (not (empty? (rest alod)))
         (cons? alod)) (build (numbers (rest alod)))]
   (list (numbers (rest alod))))))
\end{verbatim}

Now, let’s find the recurrence relation for \texttt{numbers}, where \(H(n)\) represents the recurrence relation for \texttt{build} (NOTE: \texttt{build} is not a recursive procedure!):

\begin{align*}
H(n) &= E \\
C(0) &= A \\
C(1) &= B \\
C(n) &= D + H(n) + C(n - 1) \\
 &= D + E + C(n - 1) \\
 &= Q + C(n - 1), \text{ for } n > 1
\end{align*}

Finally, through plug-n-chug we get:

\begin{align*}
C(n) &= Qn + R
\end{align*}

From this, we gather that this approach runs in \(n\) time! Hooray!

The takeaway:

• Adding a helper procedure can let us "save" a partial result and use it twice, rather than recompute it
• What makes this work is the rules of evaluation: we evaluate the actual argument before binding it to the formal argument.

• This “saving” of a partial result took us from an exponential-time algorithm to one that’s linear-time!

• BUT this is a method to use sparingly

3 Censor

Now we’re going to take a look at another silly example – one that takes in a list of strings and replaces each string with an asterisk, “*”. This is relatively straightforward, and very very similar to examples we’ve seen in the past:

\[
\text{(define (censor alos)}
\begin{align*}
& \text{(cond} \\
& \quad \text{[(empty? alos) empty]} \\
& \quad \text{[(cons? alos) (cons "*" (censor (rest alos)))])}
\end{align*}
\]

But what if we changed this up a little bit... Instead, let’s create a procedure fancy-censor that replaces each string with “*” or “+”, alternating between the two and starting with “*”. A recursive diagram would be helpful right about now!

Orig. input: (list "a" "b" "c" "d")
Rec. in: (list "b" "c" "d")
Rec. out: (list "*" "*" "*")

Orig. output: (list "*" "*" "*"")

It looks like we can add a “+” onto the end of the list, but to be sure, let’s try another example:

Orig. input: (list "a" "b" "c")
Rec. in: (list "b" "c")
Rec. out: (list "*" "+"")

Orig. output: (list "*" "+")

Now it looks like we should be adding a “*” onto the end! This problem is not as straightforward as we thought.

Instead, let’s use a helper procedure, fancy-helper:

\[
;; \text{fancy-helper: (string list) * string -> (string list)}
;; \text{Input: alos, a list of strings}
;; \quad \text{start, either "*" or "+"}
;; \text{Output: a list of alternating "*" and "+" of the same length as alos,}
;; \quad \text{starting with whichever start specifies}
;; (define (fancy-helper alos start)\]


Now, we can call `fancy-helper` in `fancy-censor`:

```scheme
;; fancy-censor: (string list) -> (string list)
;; Input: alos, a list of strings
;; Output: a list of alternating "*" and "+" of the same length as alos,
;; starting with "*"
(define (fancy-censor alos) (fancy-helper alos "*"))
```

So, by using a helper that employs an extra argument, we were able to simplify our problem.

## 4 Higher Order Procedures

Let’s consider a function that improves a list of integers by changing every element of the list to 17.

```scheme
;; improve: (int list) -> (int list)
;; Input: a list of integers, aloi
;; Output: a list of integers such that all integers
;; in aloi have been changed to be the number 17
(define (improve aloi)
  (cond
   [(empty? aloi) empty]
   [(cons? aloi) (cons
      17
      (improve (rest aloi)))]))
```

We’ve seen functions of this general format a lot: functions that do `something` to every element of a list. It turns out that this type of function is so common, that there is a much shorter way to write them:

```scheme
(map proc list)
```

Where `proc` is the procedure to apply to every element of the list, and `list` is the list that you want to change.

The type signature for `map` is something new - this is the first time that we’ve used a procedure as input. It looks like this:

```scheme
; map: ('a -> 'b)* ('a list)-> ('b list)
```

In words, this says that the arguments to `map` are: 1) a procedure with input of `a` data and output
of 'b data, and 2) a list of 'a data (the same data type as the input to the procedure). The output of map is a 'b list (the same data type as the output of the procedure.)

Map is known as a higher order procedure (HOP) because one of its arguments is a function.

5 Lambda

We now introduce a new bit of syntax in Racket: lambda. A lambda-expression evaluates to a closure. That is to say, informally, lambda-expressions are a way to produce procedures without using a define.

The syntax of a lambda-expression is:

\[(\text{lambda} \ \text{name-list} \ \text{body})\]

When evaluated, this produces a user-defined procedure (i.e., a closure), with the name-list as the argument list, and body as the body.

Here's a typical one:

\[(\text{lambda} (x) (+ x 1))\]

If we wanted to apply our user-defined procedure to a variety of integers to see what it would output, we'd do something like,

\[(((\text{lambda} (x) (+ x 1)) 5)\]
\[\Rightarrow 6\]
\[(((\text{lambda} (x) (+ x 1)) 17)\]
\[\Rightarrow 18\]

This produces a closure whose argument-list is \(x\) and whose body is \((+ x 1)\).

More generally, a lambda-expression may have many arguments and any expression at all as its body, so you can write

\[(\text{lambda} (x \ y) (* (+ x y) (- x 2)))\]

for instance.

Recall the \texttt{improve} function from the last section. Now, using map and lambda, we can rewrite the procedure like so:

\[(\text{define} (\text{improve} \ \text{aloi})\]
\[\quad (\text{map} (\text{lambda} (x) 17) \ \text{aloi}))\]

To break this down: using \texttt{lambda}, we're creating a procedure that for any input \(x\), the output is 17. Then, we're applying that function to every element of \texttt{aloi}. So, the output of \texttt{improve} is \texttt{aloi}, with every element replaced with 17.
And now the secret from earlier in the semester:

\[
\text{(define } (f \ x) (+ x 1))
\]

is really just syntactic sugar for

\[
\text{(define } f \ \text{(lambda} \ (x) (+ x 1)))
\]

The things produced by lambda-expressions are called “anonymous” because they don’t have names. They might seem useless now, but soon they’ll be your best friends.

Input: a list of string, (define (hello alos) (map (lambda (s) hello) alos)) What does this output?

6 Summary

Ideas

- “Saving” an already-computed value by means of a helper procedure can drastically reduce runtime.
- Adding an argument and/or employing a helper can make a problem more manageable.
- Higher order procedures (HOPs) are procedures that consume a function, rather than atomic or compound data.

Skills

- You now have some tools for simplifying complex problems.
- You now know how to create a recurrence relation for a procedure using the analysis recipe.

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).