CS135 Final Exam Help Session
Byron Weber Becker
2019-12-08

cons VS. append VS. list

cons/append/list: these three still confuse me quite a bit when the code gets complex. For example, recursing on

;;; weird-function: (listof (listof X)) -> Any
(define (weird-function lst)
  (append (list (first list))
          (weird-function (rest list))))

Also weird stuff like these:

(cons empty empty)
(list empty empty)
(cons (list 1 2 3) (list 1 2 3))

More cons VS. append VS. list

Why is it that (append (list X) empty) produces (list X), and (cons (list X) empty) produces (list (list X))?  

Also, why is it that you cannot append the empty list to a non-list, but you can append it to a normal list and produce the same list?

Contracts!

* cons: Any (listof Any) -> (listof Any)
* append: (listof Any) (listof Any) -> (listof Any)
* list: Any ... -> (listof Any)
### Lists

<table>
<thead>
<tr>
<th>(append</th>
<th>1 2 3</th>
<th>a b  )</th>
<th>1 2 3</th>
<th>a b</th>
</tr>
</thead>
<tbody>
<tr>
<td>(append</td>
<td>1 2 3</td>
<td>a   )</td>
<td>1 2 3</td>
<td>a</td>
</tr>
<tr>
<td>(append</td>
<td>1 2 3</td>
<td></td>
<td>1 2 3</td>
<td></td>
</tr>
</tbody>
</table>

### Lists

<table>
<thead>
<tr>
<th>(cons</th>
<th>a</th>
<th>1 2 3</th>
<th></th>
<th>a 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cons</td>
<td>a</td>
<td>2 3</td>
<td></td>
<td>a 2 3</td>
</tr>
<tr>
<td>(cons</td>
<td>a</td>
<td>3</td>
<td></td>
<td>a 3</td>
</tr>
<tr>
<td>(cons</td>
<td>a</td>
<td></td>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>

### Lists

<table>
<thead>
<tr>
<th>(cons</th>
<th>2 3</th>
<th></th>
<th>2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cons</td>
<td>3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>(cons</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
I'm confused how come sometimes foldl and foldr produce the same result and can be used interchangeably, while other times they cannot. Is there a way to know which one to use instead of trying one and realizing I want the other? I guess I don’t fully understand what foldl and foldr are doing behind the scenes.

**ALFs – filter**

\[
\text{(filter even? '(0 1 2 3 4))}
\]

**ALFs – map**

\[
\text{(map even? '(0 1 2 3 4))}
\]
**ALFs – foldr**

\[(\text{foldr } (\text{lambda} \ (x \ rorr) \ (\text{cons} \ (* \ 2 \ x) \ rorr)) \ \text{empty} \ (0 \ 1 \ 2 \ 3 \ 4))\]

**ALFs – foldl**

\[(\text{foldl } (\text{lambda} \ (x \ rorr) \ (\text{cons} \ (* \ 2 \ x) \ rorr)) \ \text{empty} \ (0 \ 1 \ 2 \ 3 \ 4))\]

**ALFs – build-list**

\[(\text{build-list} \ 5 \ (\text{lambda} \ (x) \ (* \ 2 \ x)))\]
Use of lambda

I’m a little bit confused on the use of lambda. I understand all of the abstract list functions, except I just don’t understand what lambda is and why we should use that rather than just the functions we learned earlier in this course.

\[(\text{define } \text{foo1 } x \ y \ (\ + \ x \ y))\]
\[(\text{define } \text{foo2 } (\lambda \ x \ y \ (+ \ x \ y))\]
\[(\text{foo1 } 3 \ 4) \Rightarrow 7\]
\[(\text{foo2 } 3 \ 4) \Rightarrow 7\]

**Why?**
- Avoid coming up with yet another name.
- Create the function exactly where you need it (usually a function application).
- Avoid cluttering the namespace.

Stepping lambda

Stepping through lambda in abstract list functions.

This isn’t an issue. ALFs are built-in functions. We don’t step through or into built-in functions. Just produce their value.

Note: parameters still need to be in the simplest form. A \(\lambda\) expression is already in simplest form.

nested ALFs

In the previous assignment, there were several questions in which we were told to use nested maps. I am not entirely sure how nested maps work so can we go over that in the review?

\[(\text{define } \text{foo } \text{lst})\]
\[(\text{map } f \ (\text{map } g \ \text{lst}))\]
\[(\text{define } \text{bar } \text{lol})\]
\[(\text{map } (\lambda \ \text{lst} \ (\text{map } g \ \text{lst})) \ \text{lol})\]
Yes I agree, could we go over nested abstract list functions? Not only understanding them, but also I don’t really understand how to reason about them and how to know when you can use a nested abstract list function vs explicit recursion. Sometimes my approach to a problem works but cannot be translated into abstract list functions, yet some of my classmates seem to be able to approach the problem in a much simpler way using abstract list functions. However, even if I can understand their approach after seeing it, I don’t know how I would use logic to get to that answer on my own.

- Think in composable pieces.
- Think how information moves between the parts.
- (Comment that I can’t put in print.)
- Example: 3 consecutive instances

### ALFs: 3-in-a-row

#### Abstract function practice

This is a clicker question from tutorial. I am wondering how the code looks like?

```lisp
;; 3-in-a-row desired lochar) Determines if at least three
;; consecutive occurrences of the desired character appear in lochar.
;; 3-in-a-row: Char (listof Char) → Bool
```

- A. foldr
- B. map
- C. filter
- D. build-list

These are built-in versions of the ALFs.
We didn’t discuss them; won’t be on the exam.
Example:

```lisp
(map (lambda (x y)
    (cond [(> x y) x]
          [else y]))
    '(1 2 3)
    '(3 2 1))
```
Stepping local

Stepping through multiple nested locals according to our rules.

Stepping through locals with multiple definitions.

```
(define (foo x y)
  (local [(define a 10)
           (define x (* 2 a))
           (define (b x) (+ a x y))]
         (b x)))

(foo 1 2)
```

Stepping local (cont)

```
(local [(define x1 expl) ... (define xn expn)] bodyexp) is handled as follows:

a. x1 is replaced with a fresh identifier (call it x1_new) everywhere in the local expression. The same thing is done with x2 through xn.

b. The definitions (define x1_new expl) ... (define xn_new expn) are then lifted out (all at once) to the top level of the program, preserving their ordering.

c. When all the rewritten definitions have been lifted out, what remains looks like (local [] bodyexp'), where bodyexp' is the rewritten version of bodyexp. This is just replaced with bodyexp'.

All of this (the renaming, the lifting, and removing the local with an empty definitions list) is a single step.

Stepping local (cont)

First step:
```
(define (foo x y)
  (local [(define a 10)
           (define x (* 2 a))
           (define (b x) (+ a x y))]
         (b x)))

(foo 1 2)
```

Second step:
```
(local [(define a 10)
         (define x (* 2 a))
         (define (b x) (+ a x 2))]
        (b x))
```

Third step:
```
(define a_0 10
(define x_0 (* 2 a_0))
(define (b_0 x) (+ a_0 x 2))
(b_0 x_0)
```
Recursive calls in local

Something that I really struggle with is when you have local and then you define a constant in that local that calls the function itself. I find it hard to make recursive calls without going into a never-ending loop.

Attempt at example:

\[
\begin{align*}
\text{(define (make-something lst)}
\text{(local )}
\text{[ (define result (make-something (rest lst )))]}
\text{........})
\end{align*}
\]

I also want to know different ways the use of this can be helpful.

Recursive calls in local (cont)

\[
\begin{align*}
\text{(define (make-something lst)}
\text{(cond [ (empty? lst) ...]}
\text{[else )}
\text{ (local )}
\text{[ (define result (make-something (rest lst )))]
\text{ ..})}
\end{align*}
\]

Uses:

- Clarity: give the result a \textit{meaningful} name.
- Efficiency: avoid recomputation by using \texttt{result} more than once.
- Encapsulation: hide stuff (typically a helper function)
- Scope: reuse parameters (in a locally defined helper function)

Deriving templates

I don’t understand how to derive a template from a data definition.

Example for F17 Midterm2:

\[
\begin{align*}
\text{(define-struct quux (foo bar))}
\text{;; A Quux is one of:}
\text{;; * a Num}
\text{;; * a Sym}
\text{;; * (make-quux Bool Quux)}
\text{;; * (make-quux Quux Quux)}
\end{align*}
\]
Templates and mutual recursion

I don’t understand why a template sometimes results in two (or more!) functions.

- When two (or more) data definitions refer to each other. Examples: RSpecies and EvoEvent, BinExp and BINode
- Depending on decisions you make. quux-template could have used a helper, quux-struct-template, for part of the job.
- It’s often possible cram everything into one function (but not recommended!).

Mutual recursion: height of a tree

How to turn recursion like this into mutual recursion?

```
(define (ht tree)
  (cond
    [(number? tree) 0]
    [(empty? (rest tree))
      (add1 (ht/lst (first tree)))]
    [(> (ht/lst (first tree))
        (ht/lst (second tree)))
      (ht (cons (first tree)
             (rest (rest tree))))]
    [else (ht (rest tree))])))

(define (ht/lst lot)
  (cond
    [(number? lot) 0]
    [(empty? lot) 0]
    [else
      (max (+ 1 (ht/lst (first lot)))
           (ht/lst (rest lot))))])
```

;; A LLTree is one of:
;; * a Num
;; * a (listof LLTree)
;; requires: the list is nonempty

From the sample exam:

```
(check-expect (height 5) 0)
(check-expect (height '(1 5)) 1)
(check-expect (height '(1 (2 3 (4 5)) 6)) 3)
```

Note that the problem said you had to define two functions, height and height/list.
I would just like some effective simple methods when doing mutual recursion. This one is the trickiest for me.

I was wondering if we could go over in detail about how to come up with an mutually recursive functions. I’m confused on how to approach a question that needs to use mutual recursion.

- There is nothing special about mutual recursion. It's just helper functions that happen to use each other.
- Follow the templates.
- Pay attention to your contracts.
- If you believe that your recursive calls will do the right thing, how do you combine that answer with the data you have available to solve the current problem?

3g (A8) and/or 1b (A10), I’ve read the solution multiple times but don’t really understand it

But traversing general trees like the one evaluating arithmetic expressions on slides 21 M 11, they are not really intuitive to me, so as the questions (like from assignments) gets more complicated, I get easily lost or fall into “don’t know what to do” state. (even with the help of templates/slides).
Developing accumulatively recursive functions

I was wondering if we could go over in detail about how to come up with an accumulative recursion function. I’m confused on how to approach a question that needs to use accumulative recursion.

On needing to use accumulators:

- Sometimes for efficiency (e.g. max/list used max-so-far)
- Sometimes for convenience (e.g. a list comes out in the right order)
- Sometimes a solution breaks down naturally into a series of steps where some data accumulates (e.g. max-so-far).

I’m not aware of any problem that can only be solved with accumulators. There are lots of places where accumulators have been useful.

Finding all paths in a graph

1b (A10), I’ve read the solution multiple times but don’t really understand it.

I was hoping we could go over questions relating to finding all paths to a certain key. Whether it be in a binary tree or graph. As most people on the assignments (at least the majority I have spoken too), seemed to find this the most difficult.

- In a binary tree, there’s only one path to a given key.
- In graphs, this really boils down to how can find-route return all the paths between two points rather than just one.
- Start with the contracts.