CS 135 Winter 2018

Tutorial 10: Functional Abstraction and Generative Recursion
Review: Lambda

\[ ((\text{lambda} \ (x_1 \ldots \ x_n) \ \text{exp}) \ v_1 \ldots \ v_n) \Rightarrow \text{exp}' \]

where \( \text{exp}' \) is \( \text{exp} \) with all occurrences of the parameter \( x_1 \) replaced with the argument \( v_1 \), all occurrences of \( x_2 \) replaced with \( v_2 \), and so on.

For example, the next step here would be:

\[ ((\text{lambda} \ (x \ y) \ (\ast \ (+ \ y \ 4) \ x)) \ 5 \ 6) \Rightarrow (\ast \ (+ \ 6 \ 4) \ 5) \]
Group Problem: Stepping with Lambda

Step through the following program:

```
(((lambda (x y) (lambda (x) (* x y))) 5 6) 10)
```
Functional Abstraction: compare-largest

Write a function \texttt{compare-largest} that consumes a list of numbers \texttt{lon} and produces a function. The produced function consumes a number, and produces \texttt{true} if the consumed number is strictly greater than all the numbers in \texttt{lon}, and \texttt{false} otherwise.

\[
\left((\texttt{compare-largest} \ (\texttt{list} \ 82 \ 63 \ -74 \ 21 \ 5)) \ 87\right) \Rightarrow \texttt{true} \\
\left((\texttt{compare-largest} \ (\texttt{list} \ 82 \ 63 \ -74 \ 21 \ 5)) \ 54\right) \Rightarrow \texttt{false}
\]
Clicker Question: Functional Abstraction

Consider the following definition of `mystery`. Which of the following is the most appropriate contract for `mystery`, such that any function application obeying this contract will not produce an error?

```
(define mystery (lambda (x y)
    (cond [(= (+ x 3) (− y 4)) (* x y)]
        [else (lambda (a b c)
            (<= (abs (− (string-length a)
                (string-length b))) c))))])
```

A ;; mystery: Num Num → (Num → (Str Str Num → Bool))
B ;; mystery: Num Num → (anyof Num Bool)
C ;; mystery: Num Num → (Str Str Num → (anyof Num Bool))
D ;; mystery: Num Num → (anyof Num (Str Str Num → Bool))
E `mystery` is a constant. It does not have a contract.
Generative Recursion: partition

Write a function `partition` that consumes a non-empty arbitrary list called `lst`, and a list of positive integers called `sizes`. The numbers in `sizes` must add up to the length of `lst`. `partition` splits `lst` into sublists of lengths specified by `sizes`.

Hint: Think back to how `string→strlines` was implemented in Module 11.

\[
\text{(partition '}(a \ b \ c) \ '(3)) \Rightarrow '((a \ b \ c)) \\
\text{(partition '}(a \ b \ c \ d \ e \ f \ g) \ '(3 \ 1 \ 2 \ 1)) \Rightarrow '((a \ b \ c) \ (d) \ (e \ f) \ (g))
\]
Taken from Module 11, Slide 37:

;; list→lines/gen: (listof Char) → (listof (listof Char))
(define (list→lines/gen loc)
  (local
    [(define fline (first-line loc))
     (define rlines (rest-of-lines loc))]
    (cond [(empty? rlines) (list fline)]
          [else (cons fline (list→lines/gen rlines))])))

The two helper functions are also simpler, using structural recursion on lists.
Generative Recursion: Binary Search

In this question, we will approximate a root of a function using the **bisection method**, or **binary search**. The process is described as follows:

1. Given two values *left* and *right*, determine the midpoint *mid*.

2. Given a function *f*, compute the function value at the midpoint, *f(mid)*.

3. If *f(mid)* is sufficiently small within a given *tolerance* of zero, simply produce the value of *mid*.

4. Otherwise, examine the sign of *f(mid)*, and in the next recursive call, replace either *left* or *right* with *mid*, such that the range of the interval is halved, and there is a zero crossing within the new interval.
Generative Recursion: Binary Search

Write a function `find-root` that consumes a function `f`, two numbers `left` and `right`, and a positive number `tolerance`. The function will produce a root of the function `f` using the binary search method. You may assume the following:

- `f` is a continuous function
- `left <= right`
- either \( f(left) <= 0 <= f(right) \) or \( f(right) <= 0 <= f(left) \)

```scheme
(find-root (lambda (x) (- (sqr x) 5)) 1 3 0.1) \Rightarrow 2.25
```
Group Problem (Optional): separate

Write a function `separate` that consumes a value of any type, `sep`, and an arbitrary list. It inserts `sep` between identical items in the list. Note that `sep` should not appear in the consumed list.

You may not use explicit recursion.

\[(\text{separate } 'x ' (a b c)) \Rightarrow '(a b c)\]
\[(\text{separate } 'x ' (a b b c)) \Rightarrow '(a b \times b c)\]
\[(\text{separate } 'x ' (a a b b c c)) \Rightarrow '(a \times a b \times b c \times c)\]