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:

\[
((\text{lambda} \ (x \ y) (\text{lambda} \ (x) (\ast \ x \ y))) \ 5 \ 6) \Rightarrow (\ast (\ + \ 6 \ 4) \ 5)
\]

Group Problem: Stepping with Lambda

Step through the following program:

\[
(((\text{lambda} \ (x \ y) (\text{lambda} \ (x) (\ast \ x \ y))) \ 5 \ 6) \ 10)
\]
Functional Abstraction: compare-largest

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

\[
((\text{compare-largest } \text{list} 82 63 -74 21 5) 87) \Rightarrow \text{true}
\]
\[
((\text{compare-largest } \text{list} 82 63 -74 21 5) 54) \Rightarrow \text{false}
\]

Clicker Question: Functional Abstraction

Consider the following definition of \(\text{mystery}\). Which of the following is the most appropriate contract for \(\text{mystery}\), such that any function application obeying this contract will not produce an error?

\[
(\text{define} \ \text{mystery} \ (\lambda (x \ y) \ ((\text{cond} \ ((= (+ x 3) (- y 4)) (* x y)) \ \text{else} \ (\lambda (a \ b \ c) \ (\leq (\text{abs} (- (\text{string-length} a) \ (\text{string-length} b)))) c)))))
\]

A \(\Rightarrow\) \(\text{mystery}: \text{Num} \ \text{Num} \rightarrow (\text{Num} \rightarrow (\text{Str} \ \text{Str} \ \text{Num} \rightarrow \text{Bool}))\)

B \(\Rightarrow\) \(\text{mystery}: \text{Num} \ \text{Num} \rightarrow (\text{anyof} \ \text{Num} \ \text{Bool})\)

C \(\Rightarrow\) \(\text{mystery}: \text{Num} \ \text{Num} \rightarrow (\text{Str} \ \text{Str} \ \text{Num} \rightarrow (\text{anyof} \ \text{Num} \ \text{Bool}))\)

D \(\Rightarrow\) \(\text{mystery}: \text{Num} \ \text{Num} \rightarrow (\text{anyof} \ \text{Num} (\text{Str} \ \text{Str} \ \text{Num} \rightarrow \text{Bool}))\)

E \(\Rightarrow\) \(\text{mystery}\) is a constant. It does not have a contract.

Generative Recursion: partition

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

Hint: Think back to how \(\text{string} \rightarrow \text{strlines}\) was implemented in Module 11.

\[
(\text{partition} \ '(a \ b \ c) \ '(3)) \Rightarrow '(a \ b \ c)
\]
\[
(\text{partition} \ '(a \ b \ c \ d \ e \ f \ g) \ '(3 \ 1 \ 2 \ 1)) \Rightarrow '(a \ b \ c) \ (d) \ (e \ f) \ (g))
\]
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)

(find-root (lambda (x) (- (sqr x) 5)) 1 3 0.1) => 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.

```lisp
(separate 'x '(a b c)) ⇒ '(a b c)
(separate 'x '(a b b c)) ⇒ '(a b x b c)
(separate 'x '(a a b b c c)) ⇒ '(a x a b x b c x c)
```