Advanced Mapping, Apply, and Lambda

This **`lambda`** idea is really big—so big, actually, that it deserves its own handout. It's not until you see a good set of examples that you realize how expressive the inner function is.

Generating Power Sets

Consider the problem of generating the power set of the set $\{1,2,3\}$. The power set is the set of all subsets, and an arbitrary subset can elect to include or exclude each element. Here are the 8 sets making up the power set of $\{1,2,3\}$.

$$\{\ \},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}$$

They're topologically sorted from small to large, but this presentation doesn't illustrate the inductive structure of power sets. This one does:

$$\{\ \},\{2\},\{3\},\{2,3\}$$
$$\{1\},\{1,2\},\{1,3\},\{1,2,3\}$$

Note the first of the two rows is the power set of $\{2,3\}$. The second row is the power set of $\{2,3\}$ all over again, except that a 1 has been included in each one. There's the recursive structure for you: the power set of $\{1,2,3\}$ is really two copies of $\{2,3\}$'s power set chained together, except that the second copy prepends a 1 onto the front of each entry.

Here's a **`power-set`** implementation:

```
;;
;; Function: power-set
;; ------------------
;; The power set of a set is the set of all its subsets.
;; The key recursive breakdown is:
;;
;;       The power set of {} is {{}}.  That's because the empty
;;           set is a subset of itself.
;;       The power set of a non-empty set A with first element a is
;;           equal to the concatenation of two sets:
;;              - the first set is the power set of A - {a}.  This
;;                recursively gives us all those subsets of A that
;;                exclude a.
;;              - the second set is once again the power set of A - {a},
;;                except that a has been prepended aka consed to the
;;                front of every subset.
;;
```

```
(define (power-set set)
  (if (null? set) '(())
      (let ((power-set-of-rest (power-set (cdr set))))
        (append power-set-of-rest
                (map (lambda (subset) (cons (car set) subset))
                     power-set-of-rest)))))

#|kawa:2|# (power-set '(1 2 3 4))
(() (4) (3) (3 4) (2) (2 4) (2 3) (2 3 4) (1) (1 4) (1 3) (1 3 4) (1 2) (1 2 4)
 (1 2 3) (1 2 3 4))
#|kawa:3|# (power-set '("Nils" "Nolan" "Eric"))
(() (Eric) (Nolan) (Nolan Eric) (Nils) (Nils Eric) (Nils Nolan)
 (Nils Nolan Eric))
#|kawa:4|# (power-set '(#\a #\e #\i #\o #\u))
(() (u) (o) (o u) (i) (i u) (i o) (i o u) (e) (e u) (e o) (e o u) (e i) (e i u)
 (e i o) (e i o u) (a) (a u) (a o) (a o u) (a i) (a i u) (a i o) (a i o u)
 (a e) (a e u) (a e o) (a e o u) (a e i) (a e i u) (a e i o) (a e i o u))
```

You know an algorithm is dense when the function comment is twice as long as the implementation. ☺ Note the use of a single **`let`** binding to preserve the value of the recursively generated power set. It's because of the **`let`** statement that we reduce a doubly recursive implementation to a singly recursive one. That actually reduced the running time of the algorithm quite a bit.

Of course, the algorithm is cool in its own right. But it's particularly cool in Scheme, because Scheme allows the on-the-fly **`lambda`** to be implemented in terms of the **`car`** that was excluded from the recursive call. That's how the second half is transformed from a set that excludes the first element to one that embraces it.

Embraces. How dramatic. You're rolling your eyes, I know.

Moving on. Contrast this implementation to the version implemented in C or C++. They're both compile time languages where there size of everything needs to be decided ahead of time. In C, you work around the language and use the **`void *auxData`** of **`VectorMap`** (and you also make all of the copies yourself.) In C++, you use the function object to embed the first element inside so that **`operator()`** has access to it when it's executed (and you also make all of the copies yourself.)

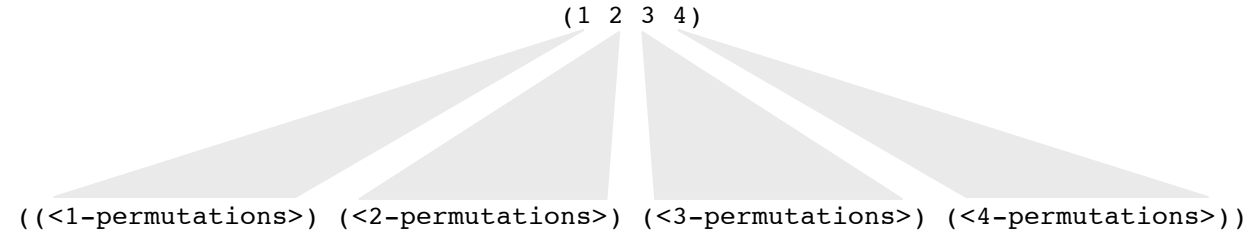
Permutations

Now consider the problem of generating all of the permutations of a list. Let's start with a simple illustration by considering the permutations of 1, 2, 3, 4. They're listed here in four columns:

1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1
1 4 2 3	2 4 1 3	3 4 1 2	4 3 1 2
1 4 3 2	2 4 3 1	3 4 2 1	4 3 2 1

Notice that the first column includes all of those permutations that start off with a 1, and that there are six of them. The second column includes the permutations starting out with a 2, similarly for columns three and four. Each column has 3! = 6 permutations, because each places some number at the front of every permutation of the other three.

What's been said so far can be framed in terms of mapping. One significant component of the map is the transformation of each element into the set of those permutations that begin with it. The mapping routine, whatever it is, needs to take an element and generate a list of permutations. Let's diagram it.



If there are four elements in the original list, there are four elements in the final list generated by any call to **`map`**. We can concatenate the four lists by **`apply`**ing **`append`** just prior to exit. That means we have a partial implementation, and it looks like this:

```
(define (permutations items)
  (if (null? items) '(())
      (apply append
         (map function-to-be-determined items))))
```

This mapping function isn't exactly trivial. It needs to transform an isolated element into a list of permutations, where every permutation in that list begins with said element. We can write it as a **`lambda`** function, which recognizes that its incoming argument is the element that needs to be at the front of all the permutations it generates. In order to generate those permutations, it must remove the element from the list, recursively

generate all of its permutations, and then map over that list so it can **cons** the distinguished element on to the front of each of its permutations. Let's write a standalone **lambda** that assumes that **items** is in scope.

```
(lambda (element)
   (map another-function-to-be-determined (permutations (remove items element))))
```

The Scheme standard doesn't require a **remove** function, and Kawa doesn't provide one either, so we need to write our own. But we certainly know what it's supposed to do, so we'll just pretend it's a built-in and worry about it later.

This inner mapping function is easier… it just needs to **cons** the incoming element to the front of each permutation produced by the recursive **permutations** call. Assuming that **element** is in scope, *another-function-to-be-determined* can be replaced by:

```
(lambda (permutation)
   (cons element permutation))
```

Now we put all of this together, to get one mean looking function. (There are dotted lines surrounding the two **lambda** functions we wrote, so you have an easier time seeing how far each one stretches.)

```
;;
;; Function: permutations
;; ----------------------
;; Generates all of the permutations of the specified list, operating
;; on the understanding that each of the n elements appears as the first
;; element of 1/n of the permutations.  The best approach uses mapping
;; to generate n different sets, where each of these sets are those
;; permutations with the nth element at the front.  We use map to transform
;; each element x into the subset of permutations that have x at the
;; front.  The mapping function that DOES that transformation is itself
;; a call to map, which manages to map an anonymous cons-ing routine
;; over all of the permutations of the list without x.  This is
;; as dense as it gets.
;;

(define (permutations items)
  (if (null? items) '(())
      (apply append
             (map (lambda (element)
                    (map (lambda (permutation)
                           (cons element permutation))
                         (permutations (remove items element))))
                  items))))

;;
;; Function: remove
;; ----------------
;; Generates a copy of the incoming list, except that
;; all elements that match the specified element in the equal?
;; sense are excluded.
;;
```

```
(define (remove ls elem)
  (cond ((null? ls) '())
        ((equal? (car ls) elem) (remove (cdr ls) elem))
        (else (cons (car ls) (remove (cdr ls) elem)))))
```

Considering you've known Scheme less than a week, you might look at the implementation of **permutations**, feel intimidated, and run back to C. Generating the set of all permutations is a difficult problem in any language. Because Scheme tries to reduce everything to a **map**, we were seemingly forced to write those **lambda**s in place without decomposition. Instead of nesting **lambda**s, we could have called a function instead, provided the function evaluates to another function that can be mapped over the elements of the list.

```
;;
;; Function: all-permutations
;; --------------------------
;; Generates all of the permutations of the specified
;; list called items.  Algorithmically identical to
;; the permutations function written above.
;;

(define (all-permutations items)
  (if (null? items) '(())
      (apply append
             (map (construct-permutations-generator items) items))))
```

This is the same as it was before, except that the big, bad **lambda** expression has been replaced by something that evaluates to that **lambda** expression. Because the original **lambda** expression was framed in terms of the local **items** parameter, we need to pass **items** to this **lambda**-generating routine. Check this out:

```
;;
;; Function: construct-permutations-generator
;; ------------------------------------------
;; Evaluates to a function (yes, a function) on one argument
;; (called element) that knows how to generate all of the permutations of the
;; specified list (called items) such that the supplied element
;; argument is constrained to be up front.
;;

(define (construct-permutations-generator items)
  (lambda (element)
    (map (construct-permutation-prepender element)
         (all-permutations (remove items element)))))
```

This is advanced functional programming, people, but we'll work you through it. Notice that the definition is equating the **construct-permutations-generator** symbol with an expression that evaluates not to a number or a string or an ordinary list, but to a **lambda** expression. That **lambda** expression itself relies on yet *another* function called **construct-permutation-prepender**, which looks like this:

```
;;
;; Function: construct-permutation-prepender
;; -----------------------------------------
;; Evaluates to a function that takes an arbitrary
;; list (presumably a permutation) and conses the
;; incoming element to the front of it.
;;
;; Notice that the implementation of the constructed
;; function is framed in terms of whatever data
;; happens to be attached to the incoming element.
;;

(define (construct-permutation-prepender element)
  (lambda (permutation) (cons element permutation)))
```

This one's easier to explain. This routine is constructing another routine whose implementation is framed in terms of the incoming **`element`**'s value. The unary routine being created is parameterized on something it chooses to call **`permutation`**, and it's equated with a **`cons`** expression that prepends **`element`** (whatever is happened to be equal to the instant the **`lambda`** is constructed) to the front of its incoming **`permutation`** argument.

```
#|kawa:2|# ((construct-permutation-prepender "where") '("did" "she" "go"))
(where did she go)
#|kawa:3|# (map (construct-permutation-prepender "vanilla")
#|(---:4|#      '(("ice" "cream") ("extract") ("latte" "with" "soy" "milk")))
((vanilla ice cream) (vanilla extract) (vanilla latte with soy milk))
#|kawa:5|# (map (construct-permutation-prepender 1)
#|(---:6|#      '((2 3 4) (2 4 3) (3 2 4) (3 4 2) (4 2 3) (4 3 2)))
((1 2 3 4) (1 2 4 3) (1 3 2 4) (1 3 4 2) (1 4 2 3) (1 4 3 2))
```

`(construct-permutation-perpender 1)` creates a unary function which prepends a 1 to the front of its lone incoming argument. Once you understand this, you can review the implementation of **`construct-permutations-generator`** to see that it constructs a function whose implementation makes reference to the value bound to the incoming **`items`**. The constructed **`lambda`** is configured to take an element (one that's presumably in the original **`items`** list) and generate all of those permutations of the original **`items`** list that just happen to have the specified element at the front.

```
#|kawa:7|# ((construct-permutations-generator '(1 2 3 4)) 3)
((3 1 2 4) (3 1 4 2) (3 2 1 4) (3 2 4 1) (3 4 1 2) (3 4 2 1))
#|kawa:8|# (map (construct-permutations-generator '(1 2 3)) '(1 2 3))
(((1 2 3) (1 3 2)) ((2 1 3) (2 3 1)) ((3 1 2) (3 2 1)))
```

This last expression gets evaluated when you type in **`(all-permutations '(1 2 3))`**. Now you see why **`append`** needs to be **`apply`**ed after the **`map`**.

Implementing map

We've already taken a shot at this, but we only managed to write the unary version of it. It wasn't an unreasonable thing to do at the time, because most **map** calls are unary anyway, and it provided an easy-to-understand example of how functions can be passed around as parameters.

The implementation of **generic-map** is intended to truly imitate the built-in. The only difference: we'll assume that all lists are of the same length, just so we can focus on implementation and not so much on error checking. Let's reproduce the implementation of **unary-map** here, because we'll be using it in a second.

```
(define (unary-map fn seq)
  (if (null? seq) '()
      (cons (fn (car seq)) (unary-map fn (cdr seq)))))
```

In order to accommodate a variable number of lists, we need to learn the Scheme equivalent of C's ellipsis. Because **generic-map** requires a function and one *or more* lists be supplied, the first line of its **define** statement needs to look like this:

```
;;
;; Function: generic-map
;; ---------------------
;; Relies on the services of the unary-map function to
;; gather the cars of all the lists to be fed the supplied
;; fn, and then gathers all of the cdrs for the recursive call.
;;

(define (generic-map fn primary-list . other-lists)
   implementation coming soon)
```

The **.** separates the required arguments from the optional ones. **fn** and **primary-list** are required, but arguments beyond **primary-list** are not. All of the additional arguments, whatever they are, are bundled together in a list called **other-lists**. When we evaluate **(generic-map * '(1 2 3) '(10 20 30) '(100 200 300))**, **fn** inherits whatever code was attached to *****, **primary-list** is tied to **(1 2 3)**, and **other-lists** is bound to **((10 20 30) (100 200 300))**.

Here's the implementation of **generic-map**, which uses **unary-map** to extract the **car**s and the **cdr**s from the lists inside **other-lists**.

```
;;
;; Function: generic-map
;; ---------------------
;; Relies on the services of the unary-map function to
;; gather the cars of all the lists to be fed the supplied
;; fn, and then gathers all of the cdrs for the recursive call.
;;

(define (generic-map fn primary-list . other-lists)
  (if (null? primary-list) '()
      (cons (apply fn (cons (car primary-list) (unary-map car other-lists)))
            (apply generic-map (cons fn (cons (cdr primary-list)
                                              (unary-map cdr other-lists)))))))
```

The first **apply** expression levies the mapping function of interest over all of the **car**s. Note that **unary-map** is used to collect the first elements of list two onward. The **car** of the first list is prepended to that so that all of the arguments are lined up like ducks in a list and the first **apply** call works.

The arguments for the recursive call are more easily assembled into a list (even the code attached to **fn** is placed in the list), and that's why **generic-map** is **apply**ed instead of being invoked by name. Constructing the list of arguments for the recursive **apply** is tricky, because all of the **cdr**s—including those of **primary-list** and within **other-lists**, need to be reestablished as peer lists, just as they were in the initial call.

```
#|kawa:2|# (generic-map cons '(1 "a" #\x) '((2 3) ("b" "c" "d") ()))
((1 2 3) (a b c d) (x))
#|kawa:3|# (generic-map * '(2 3 4) '(40 20 30) '(300 400 200))
(24000 24000 24000)
#|kawa:4|# (generic-map (lambda (fn data) (fn data))
#|(---:5}#              (list factorial fibonacci power-set permutations)
#|(---:6)#             '(6 10 (1 2 3) (a b)))
(720 55 (() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3)) ((a b) (b a)))
```