Scheme: Functions As Data

Handout written by Jerry Cain.

Consider the implementation of a **sorted?** predicate, which confirms that a sequence of numbers is sorted from low to high.

```
;;
;; Function: sorted?
;; -----------------
;; Returns true if and only if the specified list of numbers is sorted
;; from low to high.  In other words, it confirms that all neighboring
;; pairs of integers respect the <= predicate.
;;

(define (sorted? numbers)
  (or (< (length numbers) 2)
      (and (<= (car numbers) (cadr numbers))
           (sorted? (cdr numbers)))))
```

The code works just swell for numbers, but for obvious reasons won't work for strings. Take a look:

```
#|kawa:2|# (sorted? '(1 3 4 5))
#t
#|kawa:3|# (sorted? '(1 4 5 3))
#f
#|kawa:4|# (sorted? '())
#t
#|kawa:5|# (sorted? '(3.4))
#t
#|kawa:6|# (sorted? '(1/3 4/5 7/2 12/5))
#f
#|kawa:7|# (sorted? '("Alice" "Bobby" "Carol"))
Invalid parameter, was: gnu.lists.FString
java.lang.ClassCastException: gnu.lists.FString
        at gnu.kawa.functions.NumberCompare.apply2(NumberCompare.java:118)
        at gnu.kawa.functions.NumberCompare.apply2(NumberCompare.java:113)
        at atInteractiveLevel$19.isSorted(scheme-examples.scm:268)
        at atInteractiveLevel$19.apply1(scheme-examples.scm:266)
        at gnu.expr.ModuleMethod.apply1(ModuleMethod.java:191)
        at gnu.expr.ModuleMethod.apply(ModuleMethod.java:162)
        at gnu.mapping.CallContext.runUntilDone(CallContext.java:251)
        at gnu.expr.ModuleExp.evalModule(ModuleExp.java:222)
        at kawa.Shell.run(Shell.java:229)
        at kawa.Shell.run(Shell.java:172)
        at kawa.Shell.run(Shell.java:159)
        at kawa.repl.main(repl.java:744)
```

The **<=** built-in is equipped to compare numbers, but not strings (you need **string<=?** for that). We could cut and paste the **sorted?** implementation above, rename it

string-sequence-sorted?, and change the **<=** to a **string<=?**, and then we'd have a second version that works for strings.

```
(define (string-sequence-sorted? strings)
  (or (< (length strings) 2)
      (and (<= (car strings) (cadr strings))
           (string-sequence-sorted? (cdr strings)))))

#|kawa:8|# (string-sequence-sorted? '("Alice" "Bobby" "Carol"))
#t
```

Of course, cutting and pasting code is unacceptable in any language if it's at all possible to come up with an implementation that works for all cases. Fortunately, Scheme allows functions to be passed in as arguments to other function calls.

We've certainly seen it in other languages—function pointers in C and C++, function objects in C++, listeners in Java. But Scheme is just the slightest bit different. Check out the unified version of **sorted?**:

```
;;
;; Function: sorted?
;; -----------------
;; Returns true if and only if the specified list is sorted
;; according to the specified predicate function.  In other words,
;; true is returned if and only if each neighboring pair respects
;; the provided comparison function.
;;

(define (sorted? sequence comp)
  (or (< (length sequence) 2)
      (and (comp (car sequence) (cadr sequence))
           (sorted? (cdr sequence) comp))))

#|kawa:2|# (sorted? '(1 4 5 7 8 10 14) <)
#t
#|kawa:3|# (sorted? '(8.5 4.3 2.0 0.4 -1.3 -5.5) >)
#t
#|kawa:4|# (sorted? '(8.5 4.3 2.0 0.4 -7.3 -5.5) >)
#f
#|kawa:5|# (sorted? '("apple" "banana" "banana" "cherry") string<=?)
#t
#|kawa:6|# (define (list-length<? ls1 ls2) (< (length ls1) (length ls2)))
#|kawa:7|# (sorted? '((1) ("a" 'b) ((#t) 4.5 (10/3 8+7i))) list-length<?)
#t
```

On the surface, it looks like you're passing the Scheme equivalent of a function pointer, and in many ways that's true. The symbol named bound to a function implementation is actually the name attached to the code. Code isn't stored in assembly code format here—that's the C/C++ model, where a function name is understood to be the address of the function's first assembly code instruction. In Scheme, the name of a function is actually tied to the original code in list form! So, when you pass in **>** or **string<=?** or **list-length<?**, you're really passing along the code (stored as a list!) associated with

those symbols. The code is caught by a local variable called **comp**, and because **comp** sits at the front of the sub-expression:

```
(comp (car sequence) (cadr sequence)),
```

it's the symbol that evaluates to the code that tells the interpreter what to do with the remaining arguments.

Implementing mergesort

We could have framed our implementation of **partition** and **quicksort** to be generic by passing in a binary comparator function in addition to the sequence to be sorted. Rather than going back and revisiting an old example, we'll back a new sort routine by a different algorithm: **mergesort**.

Mergesort relies on a helper function—let's call it **merge**—that takes two sorted lists and synthesizes a new, sorted list by merging the original two. Mergesort is a common enough sorting algorithm that you probably saw it in CS106B or CS106X. If not, don't fret, because it's conceptually trivial.

If we have two rational number lists sorted from high to low, we can merge them this way:

```
#|kawa:2|# (merge '(1/3 1/5 1/7 1/9) '(1/2 1/4 1/6 1/8 1/10 1/11) >)
(1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9 1/10 1/11)
```

If text is your thing and you're merging string lists, you call **merge** within a string comparator instead:

```
#|kawa:3|# (merge '("c" "c++" "java" "python" "tcl") '("ruby" "scheme") string<?)
(c c++ java python ruby scheme tcl)
```

Note that the two lists don't need to be the same length. The string example above should make that clear.

The algorithm itself isn't rocket science. What makes it tough is that it's implemented in a new language. Here it is, a la Scheme:

```
;;
;; Function: merge
;; ---------------
;; Takes the two lists, each of which is assumed to be sorted
;; according to the specified comparator function, and synthesizes
;; an fresh list which is the sorted merge of the incoming two.
;;
;; If one of the lists is empty, then merge can just return the
;; other one.  If not, then the specified comp is used to determine
;; which of the two cars deserves to be at the front.  Recursion (what else?)
;; is then used to generate the merge of everything else, and the winning
;; car is consed to the front of it.
;;

(define (merge list1 list2 comp)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        ((comp (car list1) (car list2))
         (cons (car list1)
               (merge (cdr list1) list2 comp)))
        (else
         (cons (car list2)
               (merge list1 (cdr list2) comp)))))
```

The function comment explains the two base cases and the two recursive calls.

The **mergesort** routine is much easier once **merge** has been written and tested. The hardest thing about **mergesort** is figuring out how to separate the list into two lists of roughly equal length so that each can be recursively **mergesort**ed and then **merge**d together. We need a function that returns a copy of the first k elements of an n-element list. Some implementations of Scheme provide built-in functions like **nth-cdr** and (the tragically named) **butlast** to help return the front and back portions of a list. Our version doesn't, so we need to write our own quick and dirty versions. (Actually, we get away with just the second of them: I call is **prefix-of-list**).

```
;;
;; Function: prefix-of-list
;; ------------------------
;; Accepts a list and returns a new list with just the
;; first k elements.  If k is greater than the original
;; list length, the entire list is replicated.  If k is
;; negative, then don't except it to work. :)
;;

(define (prefix-of-list ls k)
  (if (or (zero? k) (null? ls)) '()
      (cons (car ls) (prefix-of-list (cdr ls) (- k 1)))))
```

```
;;
;; Function: mergesort
;; -------------------
;; Sorts the incoming list called ls so that all neighboring
;; pairs of the final list respect the specified comparator function.
;;
;; mergesort works by taking the unsorted list, generating copies of
;; the front half and the back half, recursively sorting them, and then
;; merging the two.  Classic mergesort.
;;
;; The reverse call is a bit hokey, but it brings the back half to
;; the front so that our prefix-of-list function has access to the
;; back-end elements.  The fact that the elements are in the reverse
;; of the original order is immaterial, since we don't care about
;; the original order—just the order after it's been sorted.
;;

(define (mergesort ls comp)
  (if (<= (length ls) 1) ls
      (let ((front-length (quotient (length ls) 2))
            (back-length (- (length ls) (quotient (length ls) 2))))
        (merge (mergesort (prefix-of-list ls front-length) comp)
               (mergesort (prefix-of-list (reverse ls) back-length) comp)
               comp))))
```

Notice that **front-length** and **back-length** always add up to the original list length. That's why we're careful to use **quotient** and not **/**.

It works just beautifully if we use built-ins **<**, **>** and **string>?** to sort numbers and strings:

```
#|kawa:2|# (mergesort '(1 5 3 2 5 6 3 4 5 9 2) <)
(1 2 2 3 3 4 5 5 5 6 9)
#|kawa:3|# (mergesort '(1/2 1/3 1/4 1/6 1/7 1/8 1/9
#|(---:4|#              2/3 2/5 2/7 2/9 3/4 3/5 3/7 3/8 4/5 4/7 4/9) >)
(4/5 3/4 2/3 3/5 4/7 1/2 4/9 3/7 2/5 3/8 1/3 2/7 1/4 2/9 1/6 1/7 1/8 1/9)
#|kawa:5|# (mergesort '("a" "b" "c" "d" "e" "z" "y" "x" "w") string>?)
(z y x w e d c b a)
```

And as it should, it works even if we pass in a user-defined comparator, like **distance-from-origin<?**. Look how we can sort two-dimensional points by distance from (0,0)!

```
;;
;; Function: distance-from-origin
;; ------------------------------
;; Returns the Euclidean distance of the specified
;; two-dimensional point (expressed as a list of
;; two numbers) from the origin. Thank you, Pythagorean Theorem.
;;

(define (distance-from-origin pt)
  (sqrt (+ (* (car pt) (car pt))
           (* (cadr pt) (cadr pt)))))

;;
;; Function: distance-from-origin<?
;; --------------------------------
;; Returns true if and only if the first point
;; is strictly closer to the origin that the second
;; one.
;;

(define (distance-from-origin<? pt1 pt2)
  (< (distance-from-origin pt1)
     (distance-from-origin pt2)))

#|kawa:2|# (mergesort '((3 4) (-4 3) (3 2) (6 1) (1 6) (-5 3))
#|(---:3|#              distance-from-origin<?)
((3 2) (-4 3) (3 4) (-5 3) (1 6) (6 1))
```

Let the Fun Begin: Mapping

There's a built-in routine that's so central to Scheme programming that it's imperative we learn about it and even understand how it's implemented. The function provides functionality equivalent to Assignment 3's **`VectorMap`** and the STL's **`transform`** algorithm, and it's called the strangest of thing things: **`map`**. We'll worry about its implementation in a moment. It's more important we be able to use it and use it well, so let's be the client for a second.

`map` takes a function and a list and synthesizes a new list of the same length, where each top-level element of the original has been replaced by whatever the mapping function produces when fed that element. Example time!

```
#|kawa:2|# (map car '((1 2) (3 4) (5 6) (7 8)))
(1 3 5 7)
#|kawa:3|# (map cdr '((1 2) (3 4) (5 6) (7 8)))
((2) (4) (6) (8))
#|kawa:4|# (map reverse '((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15)))
((5 4 3 2 1) (10 9 8 7 6) (15 14 13 12 11))
#|kawa:5|# (map string? '(3.4 5 1/4 "stringy" '("not" "me") "strungy"))
(#f #f #f #t #f #t)
#|kawa:6|# (map distance-from-origin '((3 4) (5 12) (-5 12) (22 121)))
(5.0 13.0 13.0 122.98373876248843)
```

All of the examples above use **map** to map a unary function over all of the elements in a single list. Turns out you can map binary functions over two lists, as with:

```
#|kawa:2|# (map append '((a b c) (x y z)) '((d e) (p d q)))
((a b c d e) (x y z p d q))
#|kawa:3|# (map + '(10 200 3000) '(7 8 9))
(17 208 3009)
#|kawa:4|# (map cons '("Haley" "Sarah" "Bond [arrogant pause]")
#|(---:5|#           '(("Joel" "Osment") ("Jessica" "Parker") ("James" "Bond")))
((Haley Joel Osment) (Sarah Jessica Parker) (Bond [arrogant pause] James Bond))
#|kawa:6|#
```

The two lists should be of equal length. The first element of the result is what it is because the **car**s of the two lists were fed, in order, to the binary mapping function. The second element of the result was produced because the **cadr**s of the two lists were paired up, and so forth. Technically, the two lists don't need to be of the same length, because **map** will just discard the extras of the longer one. We're going to pretend that's not the case and require that the two lists are of the same length.

map can take up to n lists of equal length, provided the mapping function can take n arguments.

```
#|kawa:15|# (map list '(1 2 3) '(10 20 30) '(100 200 300) '(1000 2000 3000))
((1 10 100 1000) (2 20 200 2000) (3 30 300 3000))
```

We're already in a position to implement the unary version of **map**, so let's do that. We'll call our function **unary-map** to emphasize that the mapping function can only take one argument. (We'll eventually be able to write the full version to take an arbitrary number of lists.)

```
;;
;; Function: unary-map
;; -------------------
;; We pretend that the map function doesn't exist, and we write
;; our own.  As you can see, it isn't all that difficult to get
;; a unary version of map working.
;;

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

#|kawa:2|# (unary-map list '(a "b" 4.5 (#f #t #t)))
((a) (b) (4.5) ((#f #t #t)))
#|kawa:3|# (unary-map / '(5 6 99 121))
(1/5 1/6 1/99 1/121)
```

Provided the local variable **fn** is bound to code that takes a single argument, **(fn (car seq))** generates a new element to replace the original. (Of course, the original

list isn't actually changed. It just determines the length and contents of the one being generated.)

Apply

The Scheme language provides another built-in called **apply**, which takes an n-ary function and a list of n expressions, *effectively* **cons**es the function onto the front of the argument list, and then evaluates it. Here's a trivial example:

```
#|kawa:2|# (apply + '(1 2 3 4))
10
#|kawa:3|# (sqrt (apply + (map * '(3 4) '(3 4))))
5.0
```

See what I mean when apply kinda, sorta **cons**es the **+** onto the front of **'(1 2 3 4)** to get **(+ 1 2 3 4)**, and then evaluates it as if the user actually typed **(+ 1 2 3 4)**. The **sqrt** example is yet another way to illustrate that a 3-4-5 triangle has a 90 degree angle in there somewhere.

Now, would you type in **(apply + '(1 2 3 4))** to add the first four integers? Yes, but only if you're showing off and/or the type of person who regularly recites π out to 430 decimal places for fun. But in some situations, **apply** is really the best function to use. Here's a small but meaningful example:

```
;;
;; Function: average
;; -----------------
;; Quick, clean way to compute the average of a set of numbers without
;; using any exposed car-cdr recursion.  This is the canonical example
;; illustrating why apply belongs in a functional language.
;;

(define (average num-list)
  (/ (apply + num-list) (length num-list)))

#|kawa:2|# (average '(3 5 9 11 3))
31/5
#|kawa:3|# (average '(99998 99999 100000 100001 100002))
100000
#|kawa:4|# (average '(3.4 8.3 2.7 6.6))
5.25
```

Without **apply**, we'd have to write a function to crawl down the list of numbers to produce a sum, all in spite of the fact that **+** already does something very, very similar.

Here's a real example: Let's compute the depth of a list, where the depth of the list is equal to the largest number of open left parentheses. The problem statement views a list as an ordered tree, where nested lists are viewed as sub-trees. Where there's a tree, there's a clear metaphor for depth. Let's listen in to see what **depth** has to say:

```
#|kawa:2|# (depth '("this" "list" "is" "shallow"))
1
#|kawa:3|# (depth '(this (list (has (substance)))))
4
#|kawa:4|# (depth '(1 (2 3 (4)) (((6) 7 ((((8 9)) 10) (11)))) '(4)))
7
#|kawa:5|# (depth '())
0
#|kawa:6|# (depth 4)
0
```

We could write this using **cdr-cdr** recursion, but there's a lovely implementation using **map** and **apply**. Let's say it in English first.

The depth of a tree is zero if it's the empty tree, or if it's a primitive. This explains the last two of the five sample outputs above. Otherwise, the depth of a tree is one more than the deepest of any nested sub-tree. We can compute the depth of all of the top-level elements (some of which are primitives, some of which are lists), take the largest one, and add one to it. There's a built-in function called **max** which takes one or more numbers as arguments and returns the largest one. We use it below:

```
;;
;; Function: depth
;; ---------------
;; Computes the depth of a list, where depth is understood
;; to be the largest number of open left parentheses to the left
;; of some primitive.  We assume that () doesn't appear anywhere
;; in the overall list.
;;

(define (depth tree)
  (if (or (not (list? tree)) (null? tree)) 0
      (+ 1 (apply max (map depth tree)))))
```

We're mapping a recursive function over the tree so that it's applied to every single sub-tree. Recursion doesn't get any more beautiful than this.

We can also rewrite our **flatten** function by recursively flattening all of the sub-lists, and then levying **append** to thread all of those flattened sub-lists together. This example isn't quite as elegant, because the recursive calls need to wrap primitives up as singletons so the **append** works. But it's still another good example demonstrating how nicely **apply** and **map** play together:

```
;;
;; Function: flatten-list
;; ----------------------
;; Flattens a list just like the original flatten does, but
;; it uses map, apply, and append instead of exposed car-cdr recursion.
;;

(define (flatten-list ls)
  (cond ((null? ls) '())
        ((not (list? ls)) (list ls))
        (else (apply append (map flatten-list ls)))))

#|kawa:2|# (flatten-list '(1 2 (3) 4))
(1 2 3 4)
#|kawa:3|# (flatten '(this (list (also has) substance) ((((([deep])))))))
(this list also has substance [deep])
```

Inner Function Definitions and Lambdas

Now we're going to work on a function that takes a list of numbers and generates a new list where all of the original numbers have been shifted by some offset. We'll call the function **`translate`**, and expect it to work as follows:

```
#|kawa:2|# (translate '(7 4 2 -4 8 1) 50)
(57 54 52 46 58 51)
#|kawa:3|# (translate '(1 2 3 4 5 6) 1/8)
(9/8 17/8 25/8 33/8 41/8 49/8)
```

It's simple enough to understand, but how do we implement it?

We could use **`car-cdr`** recursion, but the specification just screams **`map`**, because a list of length **`n`** is generated from another list of length **`n`**. The problem is that the mapping function required access to the delta amount—the 50, or the 1/8.

In C, we'd use the client data **`void *`** of **`VectorMap`**. In C++, we'd use a function object and embed client data inside. In Scheme, we have two options. Here's the first:

```
;;
;; Function: translate
;; -------------------
;; Takes the specified list and the specified delta and generates
;; a new list where every element of the original is shifted by the
;; specified delta.
;;

(define (translate numbers delta)
  (define (shift number)
    (+ number delta))
  (map shift numbers))
```

Note that the definition of **`translate`** defines an inner function called **`shift`**. That's something that doesn't really exist in any other language we've studied this quarter. But

Scheme (and a few modern languages—Python and Javascript come to mind) allow you to define a function within another function, on the fly. The magic is that the implementation of an inner function has **access to the set of local parameters**. That means the above implementation of **`shift`** has legitimate access not only to globally defined symbols like **`+`**, but also locally scoped ones like **`delta`**. The inner function definition only lasts as long as the outer function call, so the implementation of **`shift`** disappears as soon **`translate`** produces an answer.

There's one little functional paradigm violation here: **`translate`** is defined as a sequence of two expressions instead of just one. It's legal Scheme: the implementation of a function can be a sequence of one **or more** expressions, where the result of the last expression is taken to be the result of the entire sequence. All of the other expressions prior to the last one are executed in sequence, and the runtime only senses their side effects (like the introduction of a new function definition). We said up front that functional paradigm purists frown on side effects. But this is different, because the side effect that comes with an inner function definition is far from permanent. It lives only as long as it's needed by the outer function.

The second, more famous option is the anonymous function. If we're to use **`map`**, then translate has to map *something* over the list of numbers. That something can be defined in place as a nameless function where only the argument list and the body of the function get typed out:

```
;;
;; Function: translate
;; -------------------
;; Takes the specified list and the specified delta and generates
;; a new list where every element of the original is shifted by the
;; specified delta.  This version uses the lambda construct
;; to define a nameless function in place.
;;

(define (translate numbers delta)
  (map (lambda (number) (+ number delta)) numbers))
```

Look at that. We're calling **`map`**, and we're apparently mapping over something called **`numbers`**. Everything else, because it's parenthesized, is occupying the space that'd normally be occupied by something that evaluates to a function.

Well, that **`(lambda (number) (+ number delta))`** expression *is* a function. It's the quick and dirty way of defining a function, on the fly, without even giving it a name. Everything after the **`lambda`** symbol would be included verbatim in an inner function definition (in fact, it was included verbatim with the inner definition of **`shift`**). **`lambda`** is a gesture to programming language theory and the mathematics that backs it: lambda calculus. We invent the function definition the moment we need it. It has access to the

locally defined symbol called **`delta`**. It goes away as soon as **`translate`** is finished. No messy side effects.